Luís Miguel Picciochi de Oliveira

Licenciado em Eng. Informática

A Framework for Scientific Computing with GPUs

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : João M. S. Lourenço, Prof. Auxiliar,
Universidade Nova de Lisboa

Júri:

Presidente: Prof. Luís Monteiro, Universidade Nova de Lisboa

Arguente: Prof. Leonel Sousa, Instituto Superior Técnico

Vogal: Prof. João Lourenço, Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Março, 2011

A Framework for Scientific Computing with GPUs

Acknowledgements

First and foremost, I'd like to thank my supervisor, João Lourenço, for this opportunity, and his untiring dedication to his students' work, even when overwhelmed by huge amounts of work, never refraining from displaying his kind and humane side, strongly motivating those who work with him.

I'd like to thank Sun Microsystems for the altruistic lending of hardware with the Sun-Fire X4600 machine, to the "Problem Solving Environment for Materials Structural Characterization via Tomography" (PTDC/EIA-EIA/102579/2008) project of the Fundação para a Ciência e Tecnologia, and to professors Fernando Birra and Pedro Medeiros for their generous concessions of computing time on their multi-PU machines, that greatly enriched the experimental work described on this dissertation.

I'd like to thank Tiago Antão for sharing the Fdist application, which contributed very significantly for the validation of our framework.

I'd also like to thank my family, without whom all of my learning journey up to this day would not have been possible and to my colleagues and friends. Last but not least, an appreciation word to the countless voluntary and selfless people working on many online and offline open-source and community tech-related projects such as the Debian project, the countless collaborative communities that still find a common sharing point on multiple IRC networks, to the various members of the Unimos wireless association, among many others, to whom I am greatly in debt for the invaluable amount of knowledge that they have shared along the latest years.

This work was partially supported by the Centro de Informática e Tecnologias da Informação (CITI), and by the Fundação para a Ciência e Tecnologia (FCT/MCTES) in the research projects PTDC/EIA/74325/2006, PTDC/EIA-EIA/108963/2008, PTDC/EIA-EIA /102579/2008, and PTDC/EIA-EIA/113613/2009.

Abstract

Commodity hardware nowadays includes not only many-core CPUs but also Graphics Processing Units (GPUs) whose highly data-parallel computational capabilities have been growing at an exponential rate. This computational power can be used for purposes other than graphics-oriented applications, like processor-intensive algorithms as found in the scientific computing setting. This thesis proposes a framework that is capable of distributing computational jobs over a network of CPUs and GPUs alike. The source code for each job is an OpenCL kernel, and thus universal and independent from the specific architecture and CPU/GPU type where it will be executed. This approach releases the software developer from the burden of specific, customized revisions of the same applications for each type of processor/hardware, at the cost of a possibly sub-optimal but still very efficient solution. The proposed run-time scales up as more and more powerful computing resources become available, with no need to recompile the application. Experiments allowed to conclude that, although performance improvement achievements clearly depend on the nature of the problem and how it is coded, speedups in a distributed system containing both GPUs and multi-core CPUs can be up to two orders of magnitude.

Keywords: Job Scheduling; GPGPU; OpenCL; Run-time support; Distributed Computing; Scientific Computing

Resumo

O hardware de grande consumo para o mercado residencial já inclui não só CPUs com diversos núcleos mas também placas gráficas (GPUs), cujas elevadas capacidades de processamento paralelo têm crescido a um ritmo exponencial. Este enorme poder computacional pode ser usado em aplicações que fazem uso intensivo do processador e que não estão orientadas para o processamento gráfico, tais como alguns algoritmos comuns em ambientes de computação científica. Esta tese propõe uma *framework* vocacionada para a distribuição de trabalhos (*jobs*) computacionais numa rede de CPUs e GPUs de forma transparente para o utilizador. O código-fonte de cada *job* é um *kernel OpenCL*. Este código é universal e independente da arquitectura específica e tipo de CPU/GPU onde irá ser executado. Tal facto permite evitar a necessidade da criação de versões específicas das mesmas aplicações para cada tipo de processador/hardware. A solução obtida não será óptima para cada tipo de processador individualmente, mas em média apresentará elevados graus de eficiência. O ambiente de execução escala com a adição de novos recursos computacionais sem necessidade de recompilação da aplicação. Os resultados experimentais obtidos permitem concluir que, apesar de haver uma clara dependência na natureza do problema e da codificação da aplicação, a melhoria de desempenho observado num sistema distribuído contendo GPUs e CPUs multi-core pode ascender às duas ordens de grandeza.

Palavras-chave: Escalonamento de Trabalhos; GPGPU; OpenCL; Suporte de Run-time; Computação Distribuída; Computação Científica

Contents

List of Figures

Listings

1

Introduction

1.1 Introduction

The processing power of Graphics Processing Units (GPUs) has been climbing up on the TeraFlop (TF) scale at an exponential rate. This growth made this kind of processors very attractive for executing applications unrelated to graphics processing [OHLGSP08]. As of early 2011, with 400 USD one can buy a graphics card with a computing power of 1.35 TF. With 25 of those cards (approx. 10.000 USD), one can theoretically achieve a computing power equivalent to the last 100 clusters of the TOP 500 list of supercomputers [Top].

The computational power provided by GPUs has been used by many types of applications with High Performance Computing (HPC) requirements such as those found under the scientific computing field. Very meaningful performance improvements have been achieved but, most of the times, this has been done at the cost of an hardware-centric approach. Applications are usually developed and optimized for specific kinds of processing units and even specific models, and the implemented optimizations are hugely dependent on that hardware. Supporting tools could help in this process by providing hardware-independent performance improvements, but current such tools are not yet available and widespread. Aggregation of multiple machines is one known approach by which further performance improvements can be attained. By utilizing the processing power of multiple CPUs and GPUs, the time needed for processing specific algorithms can be reduced to a great extent.

The current trend of cluster management systems and associated job scheduling tools is to only consider CPUs as eligible processors and GPUs as mere I/O devices. GPU integration middlewares have focused on strictly local-machine integration or, for distributed systems, have only provided means for utilizing GPUs on remote machines. To

the best of our knowledge, there is a lack of job dispatching and scheduling tools for these latter systems. Load balancing on these systems could allow for optimized use of the hardware, and a transparent utilization of these heterogeneous resources to the user.

This dissertation presents a framework that considers GPUs as first-class processors. The aggregated computing power of various CPUs and GPUs on multiple computational nodes significantly improves execution times with reduced complexity for the application programmer. Some scheduling algorithms specifically designed to such environments are provided. A prototype implementation is described and evaluated with both a synthetic application and a real-world, scientific computing application. These applications were used to benchmark the framework and measure the influence of using CPUs and GPUs in their overall execution time.

1.2 Motivation

Many companies and research organizations are confronted with computational problems that require huge amounts of processing power to be solved. These HPC problems are commonly found under the scientific computing field as well as other fields of research. Usual causes for the long processing time of these algorithms are their inherent complexity and requirements for long-lasting computations, and huge sets of data that must be processed by specific algorithms.

One of the most common ways for dealing with the huge requirements of HPC applications is by resorting to multiple, interconnected machines under infrastructures commonly known as computer clusters. These infrastructures usually comprise hundreds or thousands of computational nodes under the management of a single organization. These are usually complex systems and require specialized human resources to be configured and maintained. Various software applications for these environments exist that aim at facilitating the task of cluster management. Associated with adequate job schedulers, these systems can be utilized by various HPC applications in order to reduce the execution times of such applications.

Another possible solution is to resort to the so-called computing *Clouds*. These are mostly on-line, Internet-based services that allow organizations to rent some computing time and storage space on remote systems, managed by third-party organizations, where their applications are processed.

With the advent of highly performant general-purpose GPUs (GPGPU), new opportunities for HPC have emerged in the last years, and some scientific applications have been ported to run on clusters of GPUs attaining considerable performance improvements [FQKYS04]. However, integration of this kind of hardware into "traditional" cluster systems currently treats them as auxiliary specialized processors, and not as general purpose processors capable of executing the same applications as CPUs.

Scientific computing and HPC applications in general are usually comprised of a series of steps, each one contributing to reach an intended, final solution. It is frequent

that among these steps, only a few are very processor-intensive and have HPC requirements. On these intensive steps, it is common that a huge set of data can be partitioned into smaller chunks where each chunk is processed by the same or different algorithms, respectively under a SIMD or MIMD execution model, as defined by Flynn's taxonomy [Fly72]. The common approach is to distribute these chunks of data to a set of machines, where the algorithm is processed in parallel on the available CPUs and thus reducing the program's overall execution time.

Examples of such HPC applications are very common. Some algorithms used by chemistry researchers require huge amounts of data to be processed by relatively simple algorithms: this is the case of protein docking algorithms [KKSEFAV92], useful for studying the structure of complex proteins. Other algorithms, such as gene identification algorithms [BN96] are used to identify genes associated with characteristics of animal populations —- although the most processor-intensive parts of the algorithm may not depend on huge sets of data, great amounts of time can be required in order to simulate considerable numbers of genes. Other examples not confined to the research field also follow a pattern that may allow for high parallelism: the Range-Doppler Algorithm [Esa] is one such example that is used by the military and for geological studies — given large sets of data acquired by orbiting radars and pertaining to the Earth's surface, the algorithm is used to render graphical images of the Earth's surface.

The cluster computing approach is not viable for many companies and research organizations due to its deployment and management complexity and their associated costs. On the other hand, the Cloud computing solution, although freeing the organization from the intricacies of system management, still involves high and sometimes unaffordable costs for many organizations, and introduces additional security considerations as well. When resorting to GPU computing, huge speedups might be attained, but these are usually dependent on the specificities of certain hardware models.

This thesis proposes a distributed system capable of combining the performance gains of parallel, multi-CPU execution with the huge speedups attainable by the addition of GPUs for general-purpose computations. By considering GPUs as first-class processors, hand-in-hand with CPUs, the aggregated power of multiple CPUs and GPUs provides a way to achieve high performance improvements at reduced costs. Taking into account the hardware characteristics of GPUs — massively data-parallel multiprocessors designed for the SIMD execution model —, applications that follow the SIMD execution model are among those that can make the most out of these processors. They should, preferably, be allocated to these types of processors. Applications that follow other execution models, such as MIMD, can also be made to fit into current GPUs, but with very limited performance benefits (if any). Their execution on CPUs is usually more appropriate. The framework is capable of further reducing the overall execution time by processing parts of the intended computations on less-fitting hardware, when such decisions lead to performance improvements.

The proposed system can be described as a general-purpose, GPU-aware computing framework. It resorts to OpenCL [Mun10], a platform-independent framework for writing programs capable of taking advantage of parallel architectures, including CPUs, GPUs, and other processors. The framework receives jobs submitted by applications and dispatches them for execution on an available Processing Unit (PU). Once computed, job results are returned back to the application. In order to decide to which PU should each job be submitted, the framework takes into account various aspects that differentiate these PUs such as processing power, available memory (RAM) and the bandwidth available to transfer data to be processed on the different PUs.

Classic job scheduling algorithms used by distributed job scheduling frameworks usually consider all available machines to present if not equal, at least similar computing processing capabilities. On some cases, GPUs display performance speedups over 100× relatively to regular PCs. The algorithms used on traditional clusters are not adequate to environments where processors differ so much. As such, a set of scheduling algorithms were developed for use with the proposed framework. The proposed scheduling algorithms were evaluated and benchmarked by different applications. Such experiments are described in Chapter 4, where a corresponding analysis of the attained results is also presented.

1.3 Contributions

This dissertation proposes a generic framework that allows the submission and management of jobs capable of running on CPUs, GPUs and other multiprocessor architectures. This document analyses, describes an existing implementation and evaluates the possibilities of such a system. As a result, the following contributions are given:

- The design and implementation of a generic framework for parallel job scheduling on CPUs, GPUs and other heterogeneous architectures;
- An API for easing the development and integration of domain-dependent applications willing to use the framework;
- A set of scheduling algorithms to be used with the framework, to provide appropriate dispatching of jobs to the available computing resources;
- A comparative analysis on the influence of the selected job scheduling algorithms, as well as with applications of different natures and the resulting, consequent execution times attained under each configuration;
- A study on the impact of using GPUs as first-class processors in the HPC setting.

1.4 Structure of This Document

The remainder of this document is structured as follows. Chapter 2 gives an overview over the existing technology upon which this work is made possible. In Chapter 3 we present the proposed general-purpose scientific computing framework, describing in detail its design and architecture, and each component of the system. Chapter 4 presents a validation and performance evaluation of the implemented system prototype. Finally, in Chapter 5 some concluding remarks are presented and some guides for future work are outlined. Two appendices are included: in Appendix A the API services, available for application developers for integration with the framework, are presented. Appendix B includes a set of detailed charts pertaining to the values achieved by the execution and performance tests of the experimental validation chapter.

2

Related Work

2.1 Introduction

In order to better understand the building blocks for this thesis and the proposed framework, it is essential to be aware of the relevant theory and existing technologies that enable the feasability of the proposed solution.

The main concepts of parallel and distributed computing are presented on Section 2.2. Section 2.3 covers current approaches for HPC such as cluster architectures, grids and Clouds, common on the scientific computing field. A survey of the past of GPU computing, as well as the current approaches for GPGPU is presented on Section 2.4. The main concepts of job scheduling, and the common architectures and algorithms used in distributed scheduling systems are described on Section 2.5. Finally, an overview of existing job schedulers for HPC architectures, as well as existing frameworks that allow launching jobs on graphics processing units is presented on Section 2.6.

2.2 Parallel and Distributed Computing

Parallel computing is not a recent topic in computer science and engineering. Concurrent and parallel algorithms and processors exist since at least the 1960s [Dij65]. However, processors capable of running multiple instructions at the same time on the same physical machine did not become mainstream until recently. In fact, nowadays most consumer-grade computers being sold are multiprocessors with multiple cores and even multiple CPUs. Despite the generalization of parallel hardware (CPU), most programs are still tied to the "classical" model of sequential execution that prevailed until recently.

Using multiple processors in parallel allows for an application to reduce its execution

time. This is normally achieved by following one of the models described by Flynn's taxonomy [Fly72] — Single Instruction Single Data (SISD); Single Instruction Multiple Data (SIMD); Multiple Instruction Single Data (MISD) or Multiple Instruction Multiple Data (MIMD). A short description of each follows:

- SISD — A processor handles a single stream of instructions on a single stream of data in a sequential manner. This is the "classic" approach of monoprocessors that do not provide any parallelism.
- SIMD — A processor handles one stream of instructions on multiple streams of data at the same time, in parallel. For instance, executing the same operation over a set of elements of a vector at the same time. This is the common approach for GPUs and computational architectures such as Digital Signal Processors (DSP), designed specifically to converting analog signals into digital representations. Common CPU architectures have also adopted this approach at the instruction level, for example with Intel's SSE instruction set extensions.
- MISD — This is mostly a theoretical model in which multiple instruction streams would operate on a single stream of data at the same time. There is currently no known practical application for such approach.
- MIMD — Multiple instruction streams operate on multiple data streams in parallel. This is an important model in parallel and distributed computing where multiple processors treat multiple sets of data at the same time.

Distributed computing is an expanding field of computing where an application uses computing resources that are located on independent, interconnected machines. Multiple machines are used in order to achieve reduced execution times, lower than what would be possible if a single machine was used. Distributed systems are also considered when the cost to deploy multiple slower machines is lower than what would be necessary to spend on an equivalent faster machine.

Several issues raise from the fact that the multiple processors are more decoupled than in the single-machine parallel computing model. The interconnect network's latency and throughput, the location and accessibility of the data to be processed (shared or distributed among the processors) and the associated communication model must be taken into account by the application programmer, in order to achieve the intended objective and acceptable performance gains in a distributed system.

2.2.1 Influence of the Network

The quality of the connectivity between its multiple components is a very important aspect of a distributed system. A connection's quality can be assessed by two main measures: link throughput and link latency.

The throughput of a link is defined as the amount of data that can be transmitted per unit of time. It is often measured in Megabit/s or Megabyte/s (Mb/s or MB/s, respectively).

Link latency is the time needed for a small packet of data to transit from one end of the network to another. For common LAN or Internet links it is often measured in milliseconds (ms). Link latency can be regarded as the time it takes since the sender initiated the transfer until the receiver starts obtaining the transmitted data.

For large amounts of data, latency is usually a negligible instant, making connection throughput a more important factor. For small amounts of data, latency can be a determining factor when assessing the link's quality. These concepts are important when working with distributed systems. Take, for instance, a system in which a huge amount of data must be processed by a simple (fast) algorithm. If the connection's throughput is low it may take a long time to transmit the data from one end of the system to another. In the end, it may be faster to process this data locally, thus overturning the distributed solution. On the other hand, if small amounts of data are to be processed by a slower algorithm, the network's throughput may not be that much relevant and the execution in remote faster nodes may be rewarding.

2.2.2 Memory Models

Shared Memory Shared Memory is found on parallel applications where a given memory region is accessible by multiple program threads. Each thread accesses the shared memory space, and may read or modify it. This raises several issues, as multiple threads may concurrently modify overlapping or coincident memory regions, making the state of one or more threads inconsistent with the state of the data. This is known as the critical section problem. It must be overcome by resorting to synchronization mechanisms such as locks or semaphores. Under this model, communication between processes is commonly approached by using reserved spaces of shared memory. Each thread may store information on such space, making it available for other threads. Communication among processes hosted in different machines must be provided by other means, such as MPI, described further in this section.

There are several standards and APIs that allow for taking advantage of parallel computing with shared memory spaces. Two of the most used APIs are POSIX Threads and OpenMP.

– POSIX Threads (Pthreads) [Pth] is a standard, including an API, software developers to specify the thread behaviour on a given program in a straightforward way. Typical Pthreads implementations create threads locally, on the same machine as the parent process executes, and their execution scheduling is managed by the Operating System or a user-level runtime library. Threads may run on the same CPU core as the parent process or on any of the available cores. Memory access conflicts are frequent and, in order to avoid them, locks, semaphores and synchronized

regions are provided to the programmer by the API.

– OpenMP [Omp] is an API that facilitates shared memory programming. Using OpenMP, applications may take advantage of as many processor cores as available in a straightforward way. The highest performance gains are attainable when each OpenMP thread accesses data disjoint from the data accessed by the remaining threads. This reduces the need for synchronization and allows for the highest possible speedups. Communication between processes is achieved through the usage of shared variables. The OpenMP API is focused on local shared memory models only.

Distributed Memory Under a distributed memory model, each thread can only have direct access to its own private memory space. In order to share data, threads must resort to some communication method.

– Remote Procedure Calls (RPCs) [Gro08; Thu09] are a convenient way to allow for inter-process communication. Using RPCs an application may request the execution of a given function on another process that may be running on a remote machine. When used in the context of object-oriented programming languages, RPCs are commonly referred to as Remote Invocations. Java's Remote Method Invocation (RMI) API [Cor10] is an example of such an implementation.

– Another approach lies on the usage of message-passing protocols. These allow for applications to exchange data in a simplified way. The Message Passing Interface (MPI) is a standard actively maintained by the MPI Forum [Mpia]. MPI allows to create portable applications, where point-to-point and collective communication models are possible. Using the point-to-point communication model, one program thread sends a message to another thread. Under the collective communication model, one thread may send a single message to a set of threads associated with a specific group (referred to as *communicator*). There are currently two major MPI implementations: Open MPI [GSBCBL06] and MPICH2 [Mpib]. Although the MPI standard allows implementations to provide seamless communication between machines using different architectures (such as 32-bit machines communicating with 64-bit machines), as well as multithreaded applications to send and receive messages on multiple threads at the same time, both Open MPI as well as MPICH2 support of these features is still experimental.

Distributed Shared Memory This is a memory model where running threads may execute on physically different machines. A virtual shared data space can be accessed by multiple threads, in a similar way to what occurs under the Shared Memory model. Additionally, each thread usually has exclusive access to a private data space.

– Linda [ACG86; GC92] is a language for communication/coordination between processes. It is based on a global tuple-space that may be accessed concurrently by

multiple processes. Each process may atomically add or remove tuples from this tuple-space. Tuples are identified by a tag and have a dynamic format: each tuple is composed of a set of fields and each field may hold data of any type. This flexibility makes it easy to implement distributed data types, although possibly not in the most efficient manner. Communication between processes is straightforward: one process commits a tuple onto the shared tuple-space; a receiver process may fetch that tuple and reads its contents afterwards.

– Glenda [SBF94] is a derivative from Linda. Glenda adds the possibility to use a private tuple space, where data may be addressed to a specific process. Any process may insert data into the private tuple space of any other process, but only the owner may read it. This allows for a more direct communication method using the shared memory space.

2.3 Cluster, Grid & Cloud Computing

The demand for computational power has been increasing. Current companies and organizations have to deal with more complex and demanding applications than those of the past. Applications with HPC requirements are becoming more and more common [Bak00; Hpc; Ope].

HPC is usually approached by using multiple computers. The processing speed of certain computing tasks can be increased by resorting to Networks or Clusters of Workstations (NOW/COW). Using enough resources, the performance gains of a NOW/COW can be comparable to what is achievable with regular clusters. The reduced deployment costs are usually a determining factor that is taken into account when resorting to this solution. Another approach lies on the usage of many middle- and high-end dedicated computers. These machines can be interconnected and configured in such a way as to allow for the aggregation of their capabilities. Applications using these Clusters of computers usually treat the multi-machine system as a single, complex architecture. More recently, various organizations have worked together in order to collectively achieve greater performance gains. They do so by aggregating the computing power of their clusters, making them accessible using an inter-network — creating what are usually referred to as Grids [FK99]. Seeing business opportunities on these powerful hardware infrastructures, some companies have started making them accessible to the general public. By renting storage space and processing power, they provide services to their costumers that hide the complexity of the underlying infrastructure — thus introducing the concept of Cloud computing [FZRL08]. Following, we further describe each of these infrastructures.

COW Clusters of Workstations (COW) or Networks of Workstations (NOW) [Mor03] can be created by connecting lower-end computers, such as those found at university

campuses or companies' offices. The aggregated computing power of these networked machines can be used in a concerted way in order to accelerate the processing time of highly demanding, parallelizable applications. This approach allows to put existing resources to use, by using machines that would be idle otherwise. Under these environments, the machines of the cluster can be available for users to locally log-in and use with common desktop applications. COW systems usually account for this by excluding these machines from the pool of available resources for cluster purposes. When users log-out of the machine, it can be readmitted and jobs can be submitted to it.

Figure 2.1 depicts the logotype of the Berkeley NOW Project. It portrays with a playful analogy how a considerable number of machines with lower capacities working in a concerted way can surpass the computing power of higher-end machines.

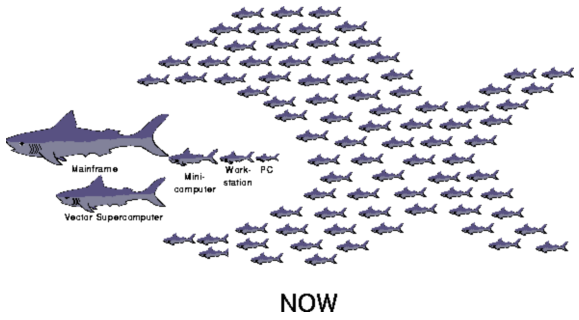


Figure 2.1: Logo of the Berkeley NOW Project. (Source: [Ber])

Computer Clusters In order to construct a specialised HPC infrastructure, an organization may resort to a computer cluster [Mor03]. Under this approach, many computers are installed on a dedicated environment and are interconnected by a high-bandwidth, low-latency network. This is in contrast to the typical mainframe approach, where all resources are centred around a single (and usually very expensive) very fast computer. To process an highly demanding task, each node of a cluster is assigned the processing of a part of the whole problem. When the program finishes running the results of the computation are gathered by resorting to some aggregation mechanism and returned to the application. Good application speedups are usually attained, although this depends on the processing speed of the machines that compose the cluster and the possibilities for parallelism presented by the application [Amd67].

The setup and management of a computer cluster is not a trivial task. The necessary facilities may not exist and may need to be built, the necessary hardware acquired and installed. The installation, configuration and maintenance of the system requires additional human resources. System administrators must manage and monitor the system on a regular basis to make sure it keeps providing acceptable levels of service. Aside from the monetary and human resources costs, this means that such infrastructures take weeks or months since the first investment is made until they are ready for production.

Moreover, with the constant progress on the processing power of new computers, organizations sooner or later will want to upgrade their hardware. Aside from the financial costs these upgrades involve, they frequently increase the complexity of the management and, consequently, the amount of work required to administer the system.

Grids It is possible for organizations to increase the potential performance of their systems without resorting to upgrades on their existing computing infrastructure. Grid computing makes this possible and is an approach that has been gaining momentum [FK99]. Different organizations cooperate by connecting their existing clusters, making their resources available to each other. Grids are distributed architectures, where each site may be administered by different organizations such as universities, governments or corporations. This makes it possible to leverage the possibilities of existing systems by aggregating their capabilities and without significant upgrades. The capabilities of such architectures are increased each time any of the involved entities improves its computing resources such as storage, computing nodes or other components, thus benefiting all participants on the grid. This allows for greater scalability, bringing more computing power available at reduced costs for each organization involved.

The distributed nature of grids poses important challenges: each grid node may be administered by a different organization and, as such, present different configurations. The heterogeneity of architectures and software systems as well as different authentication and security policies must be taken into account when planning such systems [FZRL08].

The Cloud The grid approach is still fairly inaccessible for most organizations as it usually evolves in a cooperative style: each of the involved organizations must collaborate in providing a certain amount of resources to the grid. Smaller entities are excluded from this approach as many don't have the necessary financial and/or human resources to manage such technology. Many organizations would like to be able to simply pay for some processing time and storage space on a grid so their time-consuming algorithms can run more efficiently. The so-called *Cloud* [FZRL08] is associated with this paradigm: processing time and storage are seen as a service for which the costumer pays. The costumer is granted access to these resources during the contracted amount of time and does not need to know the intricacies of the underlying hardware supporting its execution — thus the analogy with a while cloud, hiding what is on the inside.

Cloud services require the provider to ensure performance, availability and reliability at high levels so that the service can be competitive. The main Cloud service providers are organizations that can afford to serve high levels of demand to a huge number of clients without any service disruption during huge amounts of time. Companies capable of serving such services rely on resilient and greatly distributed data centres: Google, Amazon and Microsoft are some of the most well-known.

2.4 GPU Computing

There has been a constant demand from the video-game industry towards faster and more advanced graphics processors. This has lead to a very significant evolution in terms of processing power for this kind of hardware in recent years. Figure 2.2 illustrates the evolution in terms of Giga FLoating-point Operations Per Second (GFLOPS) for NVIDIA cards in comparison with the FLOPS evolution of Intel CPUs.

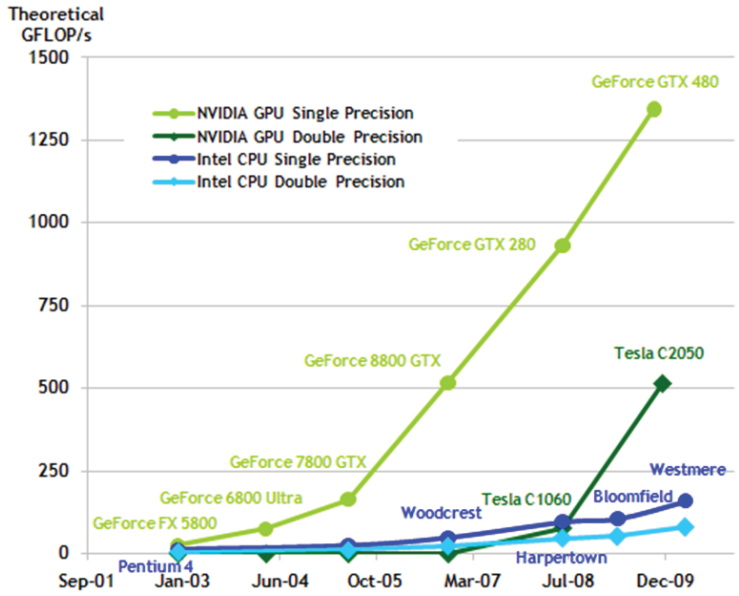


Figure 2.2: Comparison of the evolution of peak GFLOPS between NVIDIA's GPUs and Intel's CPUs. (Source: [Cudb])

The past decade saw an astonishing progress in this field: around 2002, the top-grade GPUs showed a number of FLOPS similar to the available CPUs at the time, with less than 20 GFLOPS. In 2006, GPUs were capable of performing at around 300 GFLOPS, with their CPU counterparts still under the 30 GFLOPS bar [OLGHKLP07]. As of 2011, AMD is producing graphics cards that are capable of 4,64 TFLOPS[1] and NVIDIA is reaching the 1.5 TFLOPS mark. The highest-grade CPUs are still peaking at 100 GFLOPS values.

It is important to note, however, that these numbers do not imply that a program's performance will scale proportionally to the increase in FLOPS values. Rather, they must be regarded as a sign of the evolution that this kind of hardware has been experiencing. Until recently, GPUs only outperformed CPUs in GFLOPS values for single-precision calculations. However, some GPU models already surpass the computational speed of CPUs, such as NVIDIA's Tesla C2050 general purpose computing-oriented GPU. This data allows to conclude that GPUs are presently the cheapest high performance data-parallel processors available.

The main difference between the processing capabilities of CPUs and GPUs is related

[1] ATI Radeon™ HD 5970 Graphics Specifications — http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5970/Pages/ati-radeon-hd-5970-overview.aspx#2

with their different execution models. CPUs are designed to provide high performance to sequential code; the classic approach to parallelism on a CPU was instruction-level parallelism and the main focus regarding performance was in processing a single program as fast as possible. GPUs, on the other hand, are designed to provide high processing parallelism; they are especially conceived to execute a given set of operations over huge sets of data in parallel. CPU processing is thus focused on providing low latency at the cost of lower throughputs. GPUs may display higher processing latencies but much higher processing throughputs.

Lately, CPU development has evolved to provide more processing parallelism with multiple processing cores, which drifts away from the old trend of focusing solely on increasing single-process speed [Gee05; OH05]. However, as shown above, the throughput of current CPUs is still not comparable, and is not even evolving at the same pace as the current GPUs' processing throughput.

Some models of NVIDIA's Tesla brand of GPUs do not provide any video output; others are shipped in blades ready for mounting in data-center racks — these GPUs are specifically designed for HPC, showing that the market for the GPGPU is building up and actively demanding more from this kind of processors. Also, until recently, GPUs could not be programmed in a flexible manner. This has also been changing: old GPUs could be considered as pipelines of fixed-function processors; current GPUs are better described as pipelines of programmable and fixed-function units. All of these factors create great opportunities for applications that benefit from the capabilities of modern graphics processing hardware: large computational requirements, substantial parallelism is possible, and for which computational throughput is an important factor.

In the following sections we discuss in more detail the architecture of modern GPUs, present an overview of the evolution of the programming models and languages for GPGPU, and cover the current approaches and possibilities for GPU programming.

2.4.1 Architecture of GPUs

GPUs are designed for rendering images on a screen. Although the resulting output may vary greatly (from a simple terminal emulator with plain text printed on it to a complex, high-definition, three-dimensional scene with millions of objects interacting with each other), the sequence of steps needed for rendering these images is, for the most part, very similar. In a simplified way, the typical GPU can be seen as a pipeline of steps, where the output of each step is the input of the next step.

The pipeline of a typical GPU can be explained in a simple way as a sequence of five main steps [OHLGSP08]: the initial input is a set of data representing a list of geometric primitives, formed by vertices on a three-dimensional (3D) coordinate system. The following set of operations are applied to the input data: I) *Vertex Operations*: each vertex is transformed into screen space and shaded (shadows help giving the notion of depth to three-dimensional objects) — this is an highly parallel operation as each vertex may

be processed independently of all others; II) *Primitive Assembly*: vertices are converted to triangles, the basic hardware-supported primitive of current GPUs; III) *Rasterization*: triangles are evaluated in order to determine which pixels on the 3D space they cover. Each pixel will be associated with a *fragment* primitive for each triangle that may cover it, meaning that each pixel's colour may be computed from several fragments; IV) *Fragment Operations*: each pixel is coloured according to colour information from the vertices and/or a texture residing in the GPU's global memory. This stage is also highly parallel as each pixel's computation is independent from all others; V) *Composition*: each pixel's colour is evaluated. Usually the colour of each pixel corresponds to the fragment that is nearer to the viewer or the "camera". Finally, after all the steps are computed, the rendered image is ready to be output to the screen.

Each of these stages is performed by dedicated hardware components. The typical GPU is composed of hundreds or thousands of parallel, SIMD processors, grouped according to their specialization within the graphics processing pipeline. The data to be processed is divided in chunks and input to the first processors in the pipeline; after each processing step the data is transferred onto the next specialized processors, in a succession that repeats until the final image is ready for rendering on the screen. In this way, GPUs explore both *task* and *data parallelism*: multiple stages of the pipeline are computed concurrently (*task parallelism*), as well as multiple subsets of the overall data on each stage of the pipeline (*data parallelism*). Figure 2.3 illustrates the multiprocessor architecture of a consumer-grade NVIDIA graphics card. In green and blue are different processor types; in orange, different memory regions accessible by each group of processors. Arrows indicate the possible directions of the flow of data during a processing sequence.

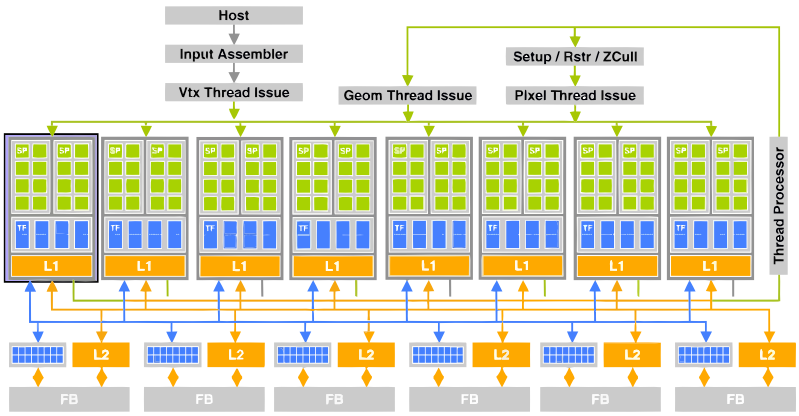


Figure 2.3: Architecture diagram of NVIDIA's GeForce 8800 GTX GPU. (Source: [OHLGSP08])

The Vertex and Fragment Operations stages described earlier were traditionally performed by fixed-function processors that the programmer could only parametrize using pre-defined options. GPUs have progressively evolved in order to allow programmers to submit customized programs to these stages. As graphical applications increased their

requirements from GPUs, GPUs have also evolved in order to provide application programmers more flexibility.

2.4.2 Evolution of General Purpose Programming on GPUs (GPGPU)

There has been significant evolution in terms of programming languages targeting the GPU. Initially, programs used the fixed-function processing units of GPUs to achieve the results they intended; as these processors progressively started to get more programmable, the first GPGPU assembly languages were introduced [OLGHKLP07].

With Microsoft's DirectX 9, higher level programming was made possible, through HLSL (High Level Shading Language) [Hls]; Cg, also a Shading Language, introduced the ability to compile to multiple targets [Cgm], and GLSL [KBR09] was a widely used, cross-platform Shading Language. All these languages were targeted at simplifying the task for programmers at rendering images, and not general-purpose programming. In order to create general-purpose programs using these languages, it was necessary to think in terms of graphics processing, and thus adapt the necessary algorithms to this paradigm, an approach that could at best be described as very hard for most applications.

BrookGPU [BFHSFHH04] and Sh [MDTPCM04] were both introduced as extensions to C that abstract the logic of the GPU and allow programming in terms of *streams*. Accelerator [TPO06] (a set of extensions to C# by Microsoft), RapidMind [McC06] and PeakStream [Pap07] allow for just-in-time compilation, and PeakStream was the first to introduce support for debugging. RapidMind was bought by Intel in 2009 and is being incorporated into Intel's in-development Ct [Rap] parallel programming technology. PeakStream was bought by Google in 2007 [Pea].

AMD also released programming languages targeting their GPUs: CTM [Hen07], which provides direct calls to the hardware while hiding the image processing logic; and CAL [Atia], with an higher-level abstraction than CTM. Later NVIDIA released CUDA [Cudb], which is an even higher-level language than CAL. By the end of 2008, the first version of OpenCL [Mun10] was released.

2.4.3 CUDA

CUDA [Cudb] is a computing architecture that enables general-purpose computing on compatible NVIDIA graphics cards. CUDA is designed to allow programming parallel applications that will scale with the evolution of future GPU architectures.

In order to run general-purpose computations on GPUs using CUDA, an application must, at some point during its execution, request the launching of a CUDA kernel on an available device. This is achieved by resorting to the CUDA API (a C / C++ library provided by NVIDIA). A CUDA kernel must be programmed in the *C for CUDA* language, a language derived from C with some additions for parallel programming on NVIDIA GPUs.

There are two main abstractions that CUDA comprises: a hierarchy of thread groups and a hierarchy of memories. These abstractions allow the programmer to think in terms of fine or coarse-grained thread parallelism as well as fine or coarse-grained data parallelism. This allows to create programs that are independent of the number of available processors, which permits them to scale independently of the hardware details and capabilities of the graphics card where they are computed.

CUDA has recently gained high visibility on the GPGPU field [Cuda] and started to be widely adopted by many applications. Driven by the market pull for massively-parallel computing, several hardware vendors started showing an high interest on GPGPU computing — leding to the creation and further development of the OpenCL parallel computing framework.

2.4.4 OpenCL

OpenCL [Mun10] is a new, open, free and platform-independent computing framework that allows software development for heterogeneous platforms. It was initially developed by Apple, and many companies such as AMD, IBM, Intel and NVIDIA later joined forces, defining the first stable specification of the language in December, 2008. Over the last two years (2009 and 2010), there have been new developments every month regarding to OpenCL, with support from new vendors, new Software Development Kits (SDKs) and new tools appearing at a steady pace. Currently it is already possible to create programs capable of running on CPUs, GPUs, and other types of multiprocessors such as (up to this date) the IBM Cell, the IBM Power VMX architecture and Creative Labs' Zii (ARM architecture) embedded systems [Ocl]. Most recently, Intel released its OpenCL implementation supporting their CPU processors.

The OpenCL architecture is heavily inspired by the CUDA model. Like CUDA, it is defined by various hierarchies of abstractions [Mun10]. These hierarchies are mapped into concepts of the *platform*, *execution* and *memory* models.

Plaftorm Model The OpenCL platform model is composed of Hosts, which are comprised of one or more Compute Devices. Each Compute Device is comprised of Compute Units, and each Compute Unit contains one or more Processing Elements. Though these abstraction are really general and implementation-dependent, these can be and are usually instantiated as follows:

- Host — A single computer;
- Compute Device — Each GPU or all CPUs in the same Host;
- Compute Unit — Each single symmetric multiprocessor of a GPU or each CPU core;
- Processing Elements — Single Instruction, Multiple Data (SIMD) elements of the Compute Units.

Execution Model An OpenCL program is composed of two main parts: the host code and the runtime code. The host code runs on the host machine's CPU and prepares the execution of the runtime code: it must select the device where the runtime will run on, input data onto the device memory and provide properties that will help the OpenCL runtime manage the task's execution. The runtime code, composed of OpenCL *kernels* (analogous to C functions) runs on the selected device, executing in accordance with the preferences previously set by the host code. The runtime code is provided by the host code in source-code format. The runtime code is compiled for the target device only in the moment when kernel execution is requested by the host code.

Each kernel is executed multiple times on the selected Compute Device. Each of these multiple instances of a kernel is called a work-item. Work-items run in parallel if the Device has the capability to do so. The number of times a kernel is executed corresponds to the total number of work-items that will be instantiated. Figure 2.4 illustrates with a visual representation the OpenCL index space for work-items. Work-items are grouped in work-groups. Work-groups (as well as work-items) are organized along a one, two or three-dimensional array which composes the global index space for all items, the N-Dimensional-Range (NDRange).

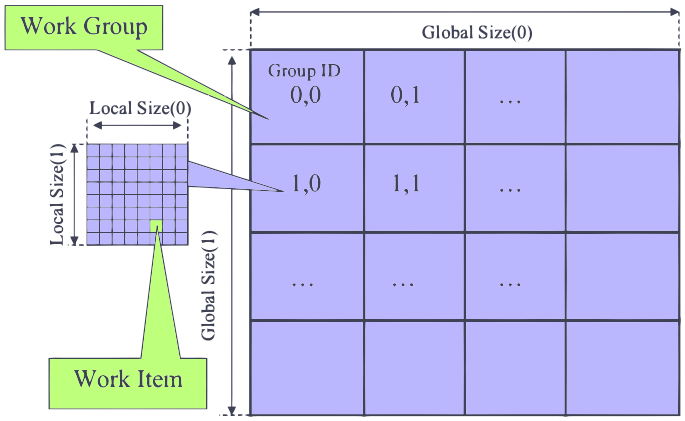


Figure 2.4: The decomposition of an OpenCL NDRange into work-groups and work-items. (Source: [Sto09])

Before launching an OpenCL program, the host code must specify the desired number of dimensions for the NDRange index space. A multi-dimensional NDRange must be used when the logical mapping of a kernel is sound with such spacial representation. The number of desired (global) work-items along each dimension of the NDRange must also be provided, as well as the number of work-items that comprise each work-group. As all work-groups must be composed of the same number of items, the total number of global work-items must be divisible by the number of items per work-group. The chosen configuration must provide an indication on the affinity between each of the items: if the application needs multiple work-items to share information, this sharing should occur as much as possible inside the same group. Every work-item is attributed a global identifier that corresponds to its N-dimensional index on the NDRange. It is also attributed a local

identifier inside its group. Every work-group is identified by an index in a similar way.

The number of total work-items as well as the number of work-items per work-group can have a huge influence on the performance of a kernel. Moreover, not all devices have the same capabilities and some may not support the same number of work-items as others. Prior to its execution, the same OpenCL kernel can (and should) be configured to fit better on the hardware it will run on. Alternatively, it is possible to choose default values recommended for the device, at the cost of a sub-optimal execution time.

Memory Model The memory model used by OpenCL also follows an hierarchical approach. It is possible to use global, constant, local or private memory. Global memory is a large, slow-access memory space, usually the DRAM on the GPU or regular RAM for a CPU — it is shared by all work-items. Constant is a read-only constant cache. Local memory allows several work-items on the same work-group to access a memory space. Private memory is only visible to each work-item. Figure 2.5 illustrates the relationship between the different memory locations and their visibility by the work-items.

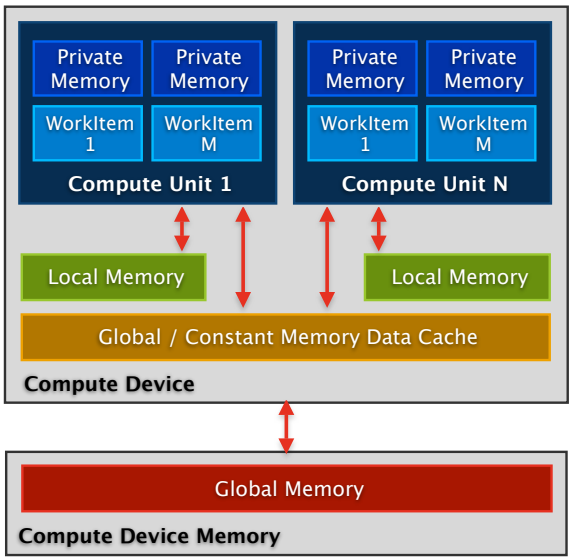


Figure 2.5: OpenCL Memory and Platform Models. (Source: [Mun08])

Given the vast diversity of hardware architectures that support OpenCL, there may be different ways to program the same kernel, which perform better on a specific kind of hardware but not on another. This means that the programmer has to decide what type of kernel is best for each target device [Sto09]. Also, the program will be more portable and faster for every architecture if different, equivalent kernels are provided for each type of device (e.g., one kernel for GPUs and one kernel for CPUs). This is a huge advantage over previous frameworks, allowing the creation of programs that may select the best algorithm to execute depending on the target architectures available at runtime.

2.5 Distributed Job Scheduling

Computer clusters are commonly used for reducing the time needed to execute applications with huge computational demands. In order to optimise the mapping of tasks to computing resources, the submitted jobs must be allocated to the available resources in an efficient way.

Non-trivial applications are usually comprised of multiple stages. Frequently, only the most processor-intensive stages of such programs are sent for execution on clusters. When submitted to a scheduler, these tasks are usually referred to as jobs. Job schedulers are, as such, an essential part of a computer cluster. They automate the resource sharing and optimize the overall utilization of the system.

Upon submission, jobs are usually inserted onto a job queue. The scheduler makes decisions taking into account various aspects of the job queue, as well as the available resources that jobs need in order to execute. Job scheduling algorithms are characterized by two main aspects [HSSY00]: the workload allocation policy and the job dispatching strategy. Workload allocation policies determine what resources shall be allocated to which jobs. For example, an allocation policy might determine that small jobs are to be distributed evenly among all available machines, all receiving the same amount of such jobs; in other instance, an allocation policy might determine that the fastest machines should be reserved for the sole execution of bigger, long-lasting jobs. Job dispatching strategies determine the order by which jobs are selected from the queue to be executed.

All jobs require a given amount of resources to be executed. It is usual for job schedulers to take these requirements into account in order to send jobs to the available resources in such a way that they are better utilized. For example, if a job requires a given number of processors to execute, it will have to wait until that number is available. Another example is when a job needs a certain amount of storage space — if that storage space is not available because it is being used by currently-running jobs, a scheduler may postpone its execution to a later time, when the required space is available.

Appropriate workload allocation policies and dispatching strategies distribute the load of the system over the available resources in a more equilibrated way, avoiding idle resources while others are overused and while providing a *fair* waiting time to job submitters until their jobs' computation results are returned to them.

Cluster Job Scheduling

A single-machine scheduler — i.e., a process scheduler like those found on common PC Operating Systems (OSes) such as Windows, GNU/Linux or MacOS — can usually achieve high usage rates. Resources are confined and the scheduler has a global view of the whole system utilization. A cluster system confined to a single site can be seen, in a simplistic way, as a single, parallel machine [HSSY00]. A central scheduler for a computer cluster can have full visibility and manageability of all available resources of the site.

These available resources are usually homogeneous and their configuration is bound to common administrative policies. Usually, network latency and throughput are neglected and are not concerns of these kinds of schedulers. Under these environments, common dispatching strategies such as *First-Come, First-Serve* (FCFS) and *Backfill* (see Section 2.5.2) are common and usually provide satisfying results.

Grid Job Scheduling

Under a grid environment, similarly to what occurs under cluster environments, programmers do not usually select the specific machines where each job will be launched. It is not humanly feasible to take into account the resource load, capabilities and the current state of each of the available sites when using a shared system with the dimensions of a grid computing system. Grid job schedulers attempt to automate this by managing the system, and scheduling each job to the most fitting site or specific computational node. The restrictions and parameters of the submitted job, as well as the available machines' current state and capabilities are usually taken into account. Aside from job schedulers capable of dispatching jobs on a grid, new algorithms and job scheduling techniques must be employed in order to handle the grid's characteristics and inherent complexity.

There has been some work on integrating existing job schedulers into grid computing infrastructures [SKSS02]. In comparison to computer clusters, grids are usually much more heterogeneous systems. Computer architectures, software environments and configurations may differ significantly across sites. Additionally, security policies may restrict access to some of the available resources. To make things even harder sites may disconnect, reconnect, or change their configurations over the course of time. The network that connects the different sites must also not be neglected — the link latency and throughput can hamper job and data transfer more significantly than on a cluster system. Deciding where to run each job becomes a much more complex task.

Given this heterogeneity as well as the possible unavailability of resources, a centralized approach is necessarily more complex and has to deal with new challenges usually not found under the cluster computing environment. Furthermore, taking into account such a huge amount of environment variability for many jobs can render the scheduler the bottleneck of the system — although, the load of the job scheduler may be alleviated, under the trade-off of producing sub-optimal scheduling decisions. The alternative is to decentralize the scheduling system. In this approach, each site may deploy different schedulers with different scheduling algorithms, potentially resulting in different overall performance results depending on the site where a job is dispatched.

Following, we present some possible architectures for distributed job scheduling and some job scheduling algorithms. Finally, we provide an overview of existing cluster and grid job schedulers as well as existing GPU-aware schedulers.

2.5.1 Scheduling architectures

Hamscher et al. [HSSY00] proposed a classification scheme for scheduler architectures. These can be applied to single-site (cluster) and multi-site (grid) scheduler architectures, although they are mostly useful in characterizing multi-site schedulers.

Centralized (Figure 2.6) One single scheduler, called the *meta-scheduler*, maintains information about *all* sites. All jobs to be executed on the system must be submitted to this meta-scheduler. The meta-scheduler dispatches jobs to the available local sites; once on the local sites, jobs are executed and no more scheduling decisions are made locally. Sites provide live information to the meta-scheduler when jobs complete and processors are free in order allow the scheduler to maintain a consistent view of the system. This approach has the downside of putting much computational weight over the single meta-scheduler. Also, the scheduler is a central point of failure, meaning that if it fails or is inaccessible for some reason none of the sites may be used for job processing, even if they are accessible and working correctly.

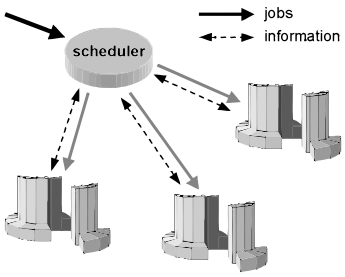


Figure 2.6: Centralized Scheduling. (Source: [HSSY00])

Hierarchical (Figure 2.7) Under the hierarchical approach, a single meta-scheduler is still responsible for receiving jobs. The meta-scheduler decides where must jobs be dispatched to. Additionally, each site has its own scheduler. There is still a single point of failure, but the load of the scheduling decisions can be more balanced and fine-grained at site-level. The scheduling algorithm used by the meta-scheduler may differ substantially from those used by the local schedulers. Also, the local schedulers may implement different scheduling algorithms between each other.

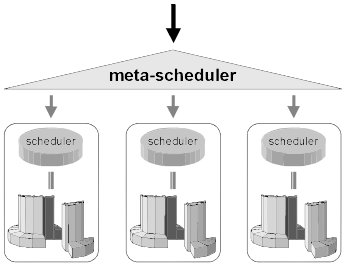


Figure 2.7: Hierarchical Scheduling. (Source: [HSSY00])

Decentralized By decentralizing the global meta-scheduler, one scheduler is used at each site and jobs may be submitted to any of them. This raises the need for some communication or synchronization mechanism between schedulers so they can share information pertaining to the current load of each site. This is a more scalable approach as the addition of more sites does not represent a significant increase on the load imposed on any of the existing schedulers. Also, the failure or inaccessibility of a single scheduler will only render its local site unavailable, and the remainder of the system may proceed operating normally. Despite these advantages, this approach may provide sub-optimal scheduling decisions as none of the schedulers has a global view of the whole system. Two alternative schemes may be adopted for optimizing this approach: using direct communication between schedulers (Figure 2.8) or a pool of waiting jobs (Figure 2.9).

- When using direct communication, schedulers may resend jobs directly to another site when a given threshold is reached. Jobs may be sent to another site with lower utilization so that they can be executed earlier than if they were kept waiting on the local queue. Schedulers also exchange information in order to maintain current information pertaining to their state so that decisions can be made using a more complete view of the system.

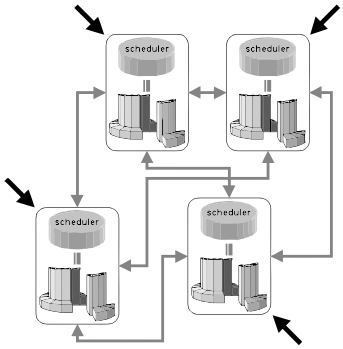


Figure 2.8: Decentralized Scheduling, resorting to direct communication. (Source: [HSSY00])

- Alternatively, jobs that can't be started soon enough due to local resource unavailability can be sent to a job-pool. When one of the schedulers detects that its resources are being under-utilized, it may fetch waiting jobs from the pool. Some balancing mechanisms may be used in order to avoid starvation, which could prevent pooled jobs from being executed — for example, weights may be attributed to pooled jobs as they get "older", ensuring that they will eventually be fetched. Alternatively, instead of being fetched from the pool, jobs may be pushed onto the schedulers at specific times.

K-Distributed Model This is a distributed scheme proposed by Viyaj Subramani et al. [SKSS02], derived from the decentralized, direct communication model. Schedulers

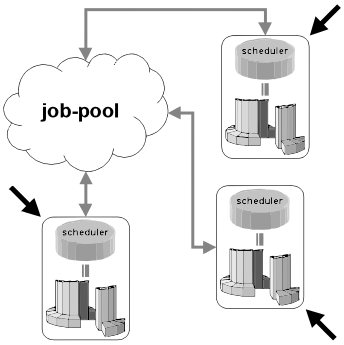


Figure 2.9: Decentralized Scheduling resorting to a Job-pool. (Source: [HSSY00])

local to each site send every job submitted to them to the K least loaded sites. When the first site initiates a job's execution, it notifies the originating site, which in turn notifies the remaining $K - 1$ sites to cancel the job in question. It has the adverse consequence that more, useless jobs will be present on all queues, a situation known as *overbooking*. This scheme also increases the communication overhead between sites.

K-Dual Distributed Model This scheme is similar to the K-Distributed model, but in this case two queues are used. One for local jobs and another for remote jobs. All jobs are sent to the K least used sites in addition to the local queue. Enqueued remote jobs are executed only when the queue for local jobs is empty or when they can backfill into the local queue without delaying *any* local jobs. In spite of this being an apparent optimization, the aforementioned K-Distributed model achieved better overall performance results than this one under the simulation environments conducted by their authors [SKSS02].

2.5.2 Job Scheduling Algorithms

Along with cluster and grid computing, job scheduling techniques and algorithms have been researched and developed at a significant pace [FR95; FRSSW97; FRS05].

Two main approaches may be followed: static or dynamic scheduling [FR95]. Static scheduling does not consider instantaneous system states when calculating job allocation decisions. The scheduler does not take into account current workload information or average system behaviours such as the mean job arrival rate at the scheduler and job execution rates.

Dynamic scheduling, on the other hand, takes the state of the system into account. Live data pertaining to the state of the components is fetched by (or pushed into) the scheduler, leading to better decisions at the cost of increased network utilization and some scheduling overhead.

Jobs submitted to a cluster or grid scheduler are usually comprised of three essential parts: the code to be executed at the computational nodes; the data to be processed by the given program; and a set of attributes. These attributes usually comprise, among other parameters, the number of processors needed by the program; input/output (I/O)

devices that may be required (such as printers, graphics cards or other devices connected to the computational nodes), as well as a prediction of the expected execution time for the given job. The first two attributes are usually seen as limiting by job schedulers: if there are not enough parallel processors or if the requested I/O devices are being used by other jobs, the job will be delayed until they become available. The predicted time, however, can be seen as simply a hint and is, most of the times, used as an upper limit for the job execution time. If a job takes longer than the predicted time, it is simply killed by the host node. Aside from expectable inaccuracies, killing jobs upon exceeding the predicted time has been reported to have the side-effect of users telling the scheduler that their jobs are much longer than they really are, in order to avoid getting their jobs killed.

Job scheduling algorithms can be categorized in two main aspects [HSSY00]: the determination of resources to be allocated for each job and the order by which jobs are selected for dispatching Allocation of resources for each job is determined by a *workload allocation policy*. Well-known allocation policies may give priority to nodes with the most free resources, nodes that have been the least utilized or follow some other policy. These are further discussed in Section 2.5.2.1. The order by which jobs are dispatched is determined by a *job dispatching strategy*. The most naïve approach is perhaps to dispatch jobs in the order by which they were submitted to the scheduler, in a First-Come First-Serve fashion. However, the impact of waiting for a job to be finished may be different for different applications. Take, for instance, the case when a user submits a job that is expected to take just a few seconds and the queue is filled with long-duration jobs. These long jobs may take hours to complete, making the short job wait until they are finished. Were the short dispatched before these long jobs, the time that the submitter would have to wait would be substantially reduced, and the impact over the long jobs might very well be negligible. Reordering of jobs may be achieved by observing the predicted execution times, by number of needed processors and/or by some categorization scheme. For example, jobs categorized as "quick administrative tasks" should be dispatched ahead of all other jobs. Jobs categorized as "non-interactive / batch" can probably go after many other smaller ones. We further discuss common dispatching strategies in Section 2.5.2.2.

2.5.2.1 Workload Allocation Policies

Workload allocation policies determine how the load is distributed among the available machines. The fraction of the total workload that each node will execute is thus determined by these policies. A *better* policy does not necessarily mean that work is more evenly distributed by every node but that the overall throughput is higher.

Some common allocation schemes are:

- BiggestFree — picks the machine with the most free resources from those that have enough resources to process the selected job. This strategy may lead to starvation because smaller jobs are allocated more easily, occupying resources needed for longer jobs, which may force them to be delayed indefinitely.

- Random — chooses a random machine from all available machines. On average, this ends up being a fair strategy, although obviously it does not optimize the overall run-time of jobs.

- BestFit — chooses the machine whose resources, upon assignment of the current job, will be utilized to their maximum extent.

- EqualUtil (also known as Dynamic Least Load) — chooses the machine that has been least utilized up to the current moment. Under heterogeneous environments this strategy has the disadvantage of not taking into account the capabilities of heterogeneous machines, wasting available resources on faster/more capable machines.

- Weighted and Optimized Weighted Workload — these are strategies proposed by Xueyan Tang et al. [TC00]. Jobs are attributed to machines using a supra-linear proportion to each machine's capabilities. Faster machines get much more jobs than slower machines. It was found out that when system load is low, it is better to assign a much greater amount of jobs to fast machines than to use a directly proportional allocation scheme. However, under heavy load and as system utilization approaches 100%, a (directly proportional) weighted scheme becomes more appropriate.

2.5.2.2 Dispatching Strategies

As discussed above, job scheduling algorithms can also be categorised based on the dispatching strategy they follow. This determines the order by which jobs are selected for execution.

Job dispatching can follow a "First-Come First-Serve" order, or jobs may be reordered. Reordering takes place in order to allow for better utilization of the resources and to provide better system throughput. Jobs can be categorised manually, by the submitting users, or by an automatic procedure. These categories can be taken into account for reordering. Pre-defined categories such as "urgent", "administrative", "non-interactive / batch" can be used. Another common categorization scheme consists of classifying jobs based on width (the number of processors they need) and length (the predicted duration). Under this approach, the available resources can be abstracted and seen as a container of fixed width, and jobs are reordered and allocated to the resources in such a way as they can fit into this width along the course of time. Figure 2.10 illustrates this abstraction.

Following, we present some of the most commonly found dispatching strategies for job schedulers:

- First-Come First-Serve (FCFS) — Jobs are sent for execution on the same order as they arrive to the scheduler. This may seem a fair policy, but it has the disadvantage of allowing big, long-running jobs go ahead of small jobs. If these smaller jobs were

executed in front of the longer jobs, the submitter would have to wait much less time for them to finish, and the difference on the bigger jobs' overall time would not be as significant.

- Shortest Job First (SJF) — Smaller jobs always go ahead of longer jobs. This benefits smaller jobs, avoiding long wait times because of much longer jobs that would be dispatched before in a FCFS and for which a smaller wait time is not significant. However, if many small jobs are submitted, this strategy can lead to the starvation of longer jobs which are never scheduled for execution.

- Random — the next job to be dispatched is picked in a (pseudo-)random way. This has overall fair results, although it is obviously not optimized for the available resources and job needs.

- Backfilling — Backfilling is an optimization over FCFS that has shown several advantages over other dispatching techniques [FRS05]. It requires each job to provide an indication of its predicted execution time. When a job at the head of the queue is waiting to be dispatched, the scheduler checks if, given the available resources, there are smaller jobs on the queue that can be dispatched before without delaying the first job. If so, they are dispatched in advance. Some issues influence the behaviour of backfilling:

- Gang scheduling — this is a more complex approach as it requires additional support from the computing environment in order to be effective. The overall idea is that currently running, less urgent or bigger jobs, can be preempted (paused) in order to give way for smaller jobs [FRS05]. This approach presents some important drawbacks. Efficiency problems arise if I/O operations are being performed by jobs that are to be preempted: these jobs will instead have to wait for the I/O operation to finish. If communication between jobs is taking place, synchronization between the affected nodes is required, implying possibly huge software overheads to keep control over such situations.

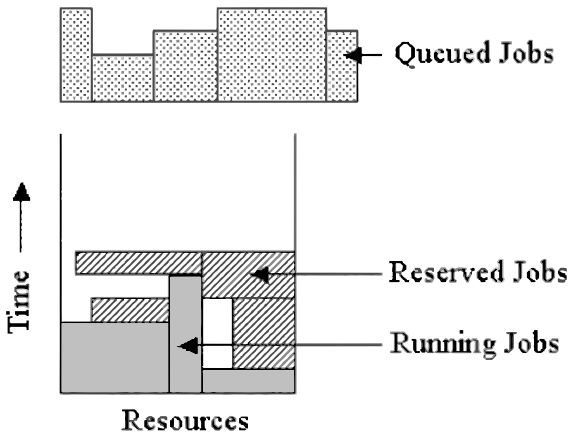


Figure 2.10: The scheduling process as a fixed-width container. (Source: [SKSS02])

- Weighted and Optimized Round-Robin — these dispatching strategies were proposed by Xueyan Tang et al. [TC00] to be used in conjunction with the Weighted and Optimized Workload policies (see Section 2.5.2.1). Jobs are dispatched to their destination machines with a proportional frequency on the amount of work attributed on each machine. This has been shown to benefit overall job execution performance.

2.6 Frameworks for High Performance Computing

There are currently many cluster and grid job schedulers, both commercial and non-commercial. Some are incorporated into more complex infrastructures that simplify the management and monitoring of cluster systems, while others are simpler programs focused on a single task. A considerable amount of work has been developed on implementing algorithms suitable to be executed on GPUs. However, supporting infrastructures and middleware systems that simplify programming and ease the task of creating programs for hybrid CPU and GPU systems are still scarce.

The following sections present an overview of the currently existing distributed job schedulers as well as GPU-aware frameworks.

2.6.1 Job Schedulers

The Moab Cluster Suite [Mab] is an integrated suite that targets to simplify the management of cluster systems. It incorporates resource monitors and many additional functionalities such as centralized management for multiple databases and storage systems. The suite's Workload Manager component dynamically adjusts workload over the nodes in a transparent way to its users. Overall, Moab allows to centralize the configuration of a single or multiple clusters, including the creation of *virtual* clusters, giving origin to even more sophisticated setups. Moab also provides some management functionalities for grid environments.

The Maui Scheduler [Mau] is a job scheduler engine that allows for job submission and management. Upon submission, it determines where each job shall be allocated according to its computational needs and the available resources. It also has some integrated system monitoring functionalities. The Maui Scheduler can be integrated into the Moab Cluster Suite.

Oracle Grid Engine [Ora] (previously known as Sun Grid Engine), although named as a *grid* engine, is mainly targeted at cluster management. Its targets are similar to those of the Moab Cluster Suite. By resorting to the Hedeby project [Hed], it is possible to integrate multi-cluster systems.

TORQUE [Tor] is a Resource Manager aimed primarily at managing available resources. It is the successor of the original Portable Batch System (PBS) [Pbs], an historically well-known scheduler. Although TORQUE has a built-in scheduler, it may be integrated with other more feature-complete schedulers.

Condor [Con] is a software framework that can be used under cluster and COW (Cluster of Workstation) environments. Its most distinctive feature is the fact that it allows the inclusion of regular workstations under a distributed computing environment. By detecting that a computer has been idle for a certain amount of time (usually two hours), it automatically makes that computer's resources available to the pool of available machines. As soon as a keystroke or the movement of the mouse is detected or if some process not managed by Condor starts using a significant amount of CPU, Condor removes the machine in question from the pool. Some integration with grid environments is also supported.

2.6.2 Local and Distributed GPU-Aware Frameworks and Schedulers

Although there is a vast array of options for computer cluster managers and schedulers, systems of this kind that take GPUs into account as first-class processors are rare, if not inexistent. Current job managers usually only take CPUs into account as job processing-capable resources, and see GPUs as mere I/O devices. A lot of interest has been shown in the past in the integration of GPUs into cluster systems in order to take advantage of distributed GPU processing power [KESSASPH09]. Different GPUs capable of general-purpose computations perform at different processing speeds, and are suited to more specific problems (preferably, SIMD tasks) than the CPU. Also, as shown in Section 2.4, the GPGPU field is evolving significantly, and the processing speed of GPUs is increasing at what appears to be an exponential rate. Current practices for GPU *integration* in clusters still do not take these important aspects into account. Following, we provide an overview of some distributed and non-distributed frameworks that take advantage of GPUs in order to improve the processing time of general-purpose applications. We also present some frameworks that allow to further extend the capabilities of computer clusters by better integrating GPGPUs into their infrastructures beyond the limiting *I/O device* approach.

A local machine-based, CPU and GPU middleware was proposed by Víctor Jiménez et al. [JVGGFN09]. It demonstrates the feasibility of a scheduling middleware that is able to dispatch jobs (or tasks) onto a GPU or CPU. The most computationally-intensive functions of the program must be submitted to the scheduler, which decides in runtime which available processor will execute the requested code segment. It resorts to CUDA, so the CPU functions have to be adapted in order to fit into the CUDA processing model. A *predictive* scheduling algorithm was developed that allows to take each PU's processing speed into account as well as the already-observed run times of each function treated by the application.

Cédric Augonnet et al. proposed a system [AN08] that is capable of providing a unified view when programming on heterogeneous (single-machine) computers comprised of both CPUs and GPUs, as well as for the Cell BE and FPGAs. The main focus of this approach was on providing a unified way for memory access on these processors, where

the memory transactions between different processors could be simplified.

In [HRFGA10], Everton Hermann et al. proposed a middleware system that aims at abstracting the number of GPUs on a single machine. Under this abstraction, a CUDA kernel is submitted to the middleware, which sends it to be processed in one of the available GPUs. If an equivalent implementation is available for both CPUs and GPUs, only one of them is executed on either one of the CPUs or one of the GPUs, as determined by a scheduler, internal to the middleware.

Maestro [SMV10] is yet another middleware framework that is designed for managing multiple OpenCL devices and optimizing resource usage by applications. It provides a single buffer for submitting data to be transferred to the devices as well as a single kernel execution request queue. Upon installation, a set of benchmarks are executed on the multiple devices in order to assert their capabilities. During kernel execution, the middleware decides on which device each kernel shall be executed. When the same kernel is submitted multiple times, Maestro performs variations upon the NDRange index space configuration, data transfer size, as well as different divisions of work among the devices in order to reduce the run-time of each kernel. As long as the kernel run time is reduced, further variations are tested.

CaravelaMPI [YS09] is a distributed computing environment that enables using multiple GPUs on a number of machines for scientific computations. Using a *flow-model*, it allows to submit an input data stream onto the framework, receiving an output data stream after it has been processed by a number of pipelined GPUs. This framework resorts to the GLSL shading language, meaning that tasks submitted to it will only be executed on GPUs.

rCUDA [DPaSMQO10] allows for remote CUDA kernel processing. No job scheduling takes place. When kernels are submitted, they are immediately dispatched to the selected device and it is left to the CUDA runtime's responsibility to manage the execution of the kernels on the local queue of each device.

MOSIX is a cluster management system [BS10b; BS10a]. It is distributed as a patch for the Linux kernel and its main target is to offload processes from machines to other, less-loaded, interconnected machines. Processes are usually started on a local machine; if, during their execution, it is determined that another computer in the cluster has enough resources as to increase the performance of the computations, the process is automatically paused, transferred and resumed on the target machine. This cluster system has recently released an OpenCL-based support layer (Virtual OpenCL layer, VCL) [BBNLS10]. This layer allows for applications to see all (configured) GPUs and CPUs installed on all machines in the cluster as if they were directly accessible from the host machine. In a similar way to the rCUDA approach, applications can then request the execution of OpenCL kernels in one or in multiple OpenCL devices.

2.7 Summary

In this chapter we presented the fundamental concepts of parallel and distributed computing. We covered the concepts and usual approaches of Cluster, Grid and Cloud computing. Afterwards, we reviewed the recent history as well as the current state of General-Purpose computing on GPUs (GPGPU), including an overview on the hardware architecture as well as past and current supporting languages for GPGPU. Finally, we covered some infrastructures used for distributed job scheduling under cluster and grid environments, as well as some common job scheduling algorithms for these distributed systems. Finally, we presented an overview on previous work on frameworks that allow for better integration of applications into heterogeneous computing systems comprising multiple CPUs and/or multiple GPUs, both under local and distributed environments.

The following chapter introduces a proposal for a distributed job scheduler for heterogeneous environments. Its design and the currently implemented prototype are presented and discussed.

3

A Distributed Computing Framework for Heterogeneous Environments

3.1 Introduction

This chapter presents a distributed computing framework designed to take advantage of computer systems comprised of heterogeneous architectures. The framework may be deployed either on a single local machine, or on multiple distributed resources, allowing programs to take advantage of all the heterogeneous Processing Units available in these systems.

Section 3.2 describes the system design, proposing a framework composed by a set of components, with well-defined roles. The role of Jobs on the framework is introduced and each of the framework's components is subsequently described in detail. An Application Programming Interface (API), designed to facilitate the integration of applications with the framework's components, is also presented in this section. The details of the prototype that was developed and implemented in order to validate the proposed system are covered in Section 3.3. The architectural configurations supported by the system are also described in this section. Afterwards, the most relevant implementation details of each of the framework's components are described, and the chapter finishes with a thorough description of the implemented scheduling algorithms.

3.2 System Design

As previously mentioned, small and medium-sized organizations that have high requirements for computing currently have to deal with serious limitations: cluster, grid and cloud computing might be too expensive and difficult to manage in order to be realistic options. Turning to the GPU solution, and as shown in Section 2.5, current ways of integrating multiple and heterogeneous computational resources on the same environment do not presently provide a seamless, consistent view to the programmer. Developers usually have to explicitly determine in development-time what Processing Unit will process each part of a program's algorithms and end up optimizing their code for these specific processors. Under distributed environments, more often than not, job schedulers do not even consider GPUs as processing units capable of processing jobs, and leave the handling of such hardware units at the consideration of the software developer.

Taking the current state of GPU and heterogeneous computing into account, a few core objectives were kept in mind while designing the framework: i) all PUs available on a computer should be considered eligible for accelerating the processing time of high-demanding programs; ii) the resulting framework should be reusable for other scientific computing and HPC applications; iii) it should be easy to adapt the framework to run under different hardware and network configurations and iv) scalability to an arbitrary number of computers and PUs should be strived.

The framework was conceived by grouping the major roles of the system into different components. The functionalities of each component are well contained, but the components must be able to communicate with each other in order to progress on their flow of execution. We make a distinction between the *domain-dependent* and the *domain-independent* components. Domain-dependent components must be programmed by or with the aid of a domain specialist who must be knowledgeable about the application and is expected to know how to interpret the results returned by the application. On the other hand, domain-independent components may be reused for other applications and computational environments. Despite allowing some customisation by appropriate configuration options, these domain-independent components do not need to be modified in order to support new applications or hardware configurations.

Applications that use the framework create Jobs, which represent the work unit that is submitted to the system for execution. The component that generates these jobs is the Job Manager (JM). Being the only domain-specific component, the JM must be designed specifically for each application willing to use the framework, and must be developed with the aid of a domain specialist. In order to simplify the integration of applications into the existing framework, the JM is linked to the application and resorts to an Application Programming Interface (API) for accessing and communicating with the remainder of the framework. Appendix A contains a description of the services made available by the API to the JM.

After creating jobs, the JM must submit them to the framework. The PU where each

job will be processed is determined by the framework, jobs are processed and, upon completion, their results are returned to the JM that created and submitted them. Figure 3.1 illustrates the job-handling workflow on the framework. It can be described as follows: 1. After creating a Job, the JM submits it to the framework. The Job Scheduler (JS) is the component responsible for receiving newly-received jobs; 2. The JS selects a PU where the Job will be executed. The Job is then sent to the corresponding Processing Unit Manager (PU-M); 3. The PU-M launches the Job onto the selected PU. Upon Job execution completion, the results of the computation are sent to a Results Collector (RC); 4. Finally, the Job's computation results are available and can be fetched from the RC by the JM.

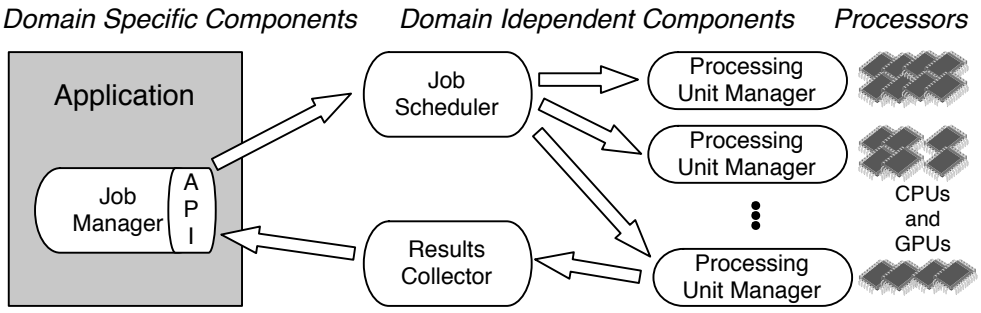


Figure 3.1: Framework main components and Job workflow.

All of these components may be running on a single node or scattered among a set of nodes. Each component is only required to be able to communicate with the components with which it must exchange data. Different instantiations of the framework may also deploy different numbers of components of each type in order to better utilize specific hardware configurations. We present some local and distributed configurations that may be used under typical COW, cluster and grid environments:

- A single instance of each component, all components on the same machine — this configuration can be used to take advantage of multiple PUs existent on a single machine. Jobs created by the Job Manager are submitted to the local Job Scheduler, which chooses a local adequate PU to execute each Job. This allows to transparently take advantage of all resources available on a single machine.

- One central Job Scheduler; multiple, distributed PU-Ms — appropriate for environments where each computational node may be comprised of multiple PUs, such environments COWs in scientific research organizations, or as in corporate computer clusters. A PU-M may be installed on each node. On the simplest case, a single Job Manager submits jobs to a Job Scheduler, which dispatches them to chosen PU-Ms. If necessary, multiple Job Managers may submit jobs to the same, central Job Scheduler. In order to distribute the data comprising job results among multiple nodes, multiple instances of Results Collectors may be deployed — in this case, RCs should preferably be deployed on machines with generous amounts of RAM and/or hard-disk storage space in order to handle all the received data.

- Multiple Job Schedulers; multiple PU-Ms — under a multi-cluster/grid environment where each node of each cluster is running an instance of a PU-M, a Job Scheduler may be installed on a central/"entry-point" node for each computer cluster. Job Managers submit jobs to one of the JSs, that dispatch these jobs to the PU-Ms on their respective, local cluster. This configuration may create some decisional effort on the Job Manager's programmer, as a decision must be taken on to which JS each job should be sent.

  One possibility to overcome the decision problem on the JM's part is to deploy one additional Job Scheduler, acting as a *meta-scheduler*[1]. Jobs are submitted to the meta-scheduler, which decides to which of the other schedulers each job should be dispatched. Subsequently, the local (cluster-level) Schedulers decide to which PU-M should jobs be dispatched to.

  Another possibility is to have a PU-M managing a set of nodes. This allows for using a single Job Scheduler, dispatching jobs to these PU-Ms. For this to be possible, an underlying OpenCL implementation should provide a unified view of a set of devices spanning multiple computational nodes to a single PU-M. Projects such as MOSIX VCL [BS10b] seem to be able to bring forth this possibility, but further research into this solution is necessary in order to assert its feasibility.

Following, we present each of the framework's building blocks in detail.

3.2.1 Jobs

Jobs are the basic computational work units handled by the framework. A Job comprises a task describing the algorithm to be executed, the input data to be processed, and a set of attributes. Figure 3.2 depicts a schematic representation of a Job.

A Job's task defines the algorithm to be executed. It must be provided as the source-code of an OpenCL kernel. Due to the heterogeneity support of OpenCL, this same kernel may be launched on different kinds of multiprocessor architectures such as CPUs, GPUs as well as other kinds of processors (see Section 2.4.4). Job attributes are: an unique identifier (ID), a category identifier, a set of Processing Unit preferences, the desired NDRange configuration for the kernel defined on the job's task and a set of arguments containing the data to be processed.

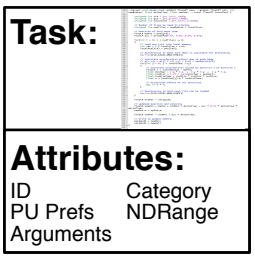


Figure 3.2: The components of a Job.

Every Job must have its own identifier (ID). An ID must be unique for every job submitted by the same JM, for at least as long as the given job is present on the framework (i.e., it has been submitted and the corresponding results were not yet collected by the originating JM).

[1] This configuration is not yet supported by our prototype.

Each Job can be associated with a user-defined category. Jobs that will predictably display a similar behaviour when processed on the same PU should be considered as jobs of the same category and defined as such by the software developer. This may be used by some scheduling algorithms to improve scheduling decisions. See Section 3.3.4 for further details on job categories.

Processing Unit preferences provide a set of informations and hints for selecting the most appropriate *type* of PU for the given Job. A PU type corresponds to a specific OpenCL-supported device type. At the time of this writing, they are: CPU, GPU or ACCELERATOR [Mun10]. There are five levels of PU preference:

- Required — only one type of PU may be defined as `Required`. The job may only be processed by that type of PU. If no PU of that type is available, the Job is not processed by the framework at all.
- Preferred — the given job will be preferably processed by PUs of the types indicated as `Preferred`. This may be used to indicate that the programmer knows that this job will perform better on some PUs than others.
- Allow — much like `Preferred` PU types, types marked as `Allow` may also be chosen by a JS to dispatch the given job. These types should be given a lower precedence when making scheduling decisions, in comparison to `Preferred` PU types.
- Avoid — the given job may be processed on PUs of type marked as `Avoid`, although the programmer would rather that not to happen. This can be used when the user expects that the algorithm on the job's task will perform bad on a given type of PU. However, if there are no alternatives or the existing ones are not satisfying, the job may be processed there as a last resort.
- Forbid — the given job may *not* be processed on PUs marked as `Forbid`. This is useful when a job's task is using certain OpenCL extensions not found on all PU types and will therefore fail on PUs that don't support them, or when the algorithm is known to perform so much worse on a given PU type that it should not be considered for processing there.

These levels must be used at the user's convenience. For example, if a given job's task consists of an intensive highly data-parallel (SIMD) computation, one will probably prefer it to be executed on a GPU. On the other hand, if the job requires the computation of a long, complex algorithm, that may branch to completely different steps of the algorithm for different input data, it may be more appropriate to request that the job be executed on a CPU.

An indication on the number of OpenCL work-items that must process the job may also be provided. Albeit an optional attribute, choosing an optimized NDRange configuration may increase job processing performance significantly. It this attribute is omitted,

the defaults given by the OpenCL implementation for the selected PU will apply at runtime.

Data to be processed must be associated with the Job's arguments. These arguments correspond to the parameters of the OpenCL kernel on the Job's task. Arguments may be of four types: input data, which will be transferred to the destination PU as provided by the JM; output data, which will contain the job's computation results and will be returned to the JM after the job is completed; empty buffers, which will be allocated in global memory for use by the kernel, but does not contain valid data at the start of the kernel and will be lost at its end. An argument may also contain both input data at the start of the kernel's computation and output data at the end of it, in which case its contents will be transferred to the PU and back again.

Jobs must also indicate the specific Results Collector from which the submitting Job Manager expects to fetch the data returned at the end of the job's execution.

3.2.2 Job Manager (JM)

The Job Manager is the system's domain-specific component. Given an HPC application, the JM is the portion of the program that is responsible for creating jobs, submitting them and receiving their computations' results from the framework.

The JM must be conceived taking into account the properties of the data to be processed and the domain-specific algorithm. Applications with high-processing needs are usually composed of multiple steps [KKSEFAV92; BN96; Esa; Aze09]. Each step may correspond, for example, to a simple calculation of a few numbers; a processing-intensive computation; or a simple, repetitive task that spans large amounts of data. Steps may or may not depend on values determined by previous steps. A JM should create jobs for those steps that take a long time to be computed in order to reduce their overall execution time. Depending on the nature of the algorithms, applications can benefit from task and/or data parallelism. Using task parallelism, multiple steps are processed simultaneously. With data parallelism, the data processed on a step is partitioned and the same algorithm is executed in parallel by multiple processors, doing the same computation over different sets of data. This is reflected on the way jobs are created. When task parallelism is intended, jobs with different tasks, each corresponding to a different stage of the algorithm, are created and submitted to the framework. When data parallelism is intended, jobs with the same task, each with a different set of data, are submitted instead.

The JM may request to be notified when the results of a previously-submitted job are available. When the notification arrives, the JM may then collect the job's output data. This mechanism makes it possible for the JM to continue processing further computations while at the same time being able to receive the requested data as soon as it is available. Alternatively, the Job Manager may query the Results Collector from time to time in order to check if the required data is available and, subsequently, fetch it. After receiving the results from a previous computation, the JM may store this data and, if needed, create

new jobs based on it.

3.2.3 Application Programming Interface (API)

To aid the development of the domain-specific Job Manager, an Application Programming Interface (API) is provided. The API can be seen as a thin layer that facilitates the integration between the domain-specific and the domain-independent components of the framework. Using the API, users of the framework and domain specialists are freed from having to think in terms of the lower-level communication layer as well as the details of the job-building process. They may instead focus on the aspects of task creation, defining job attributes, data partitioning and data gathering. Aside from facilitating the development process, the services provided by the API also ensure that no obviously invalid data is input into the system, and issue warnings or errors under such situations.

The API provides data types, structures and variables to simplify the job creation process. Functions are also provided to ease the creation of jobs, their submission to the framework, and retrieval of computation results. A complete description of the available API services and utilities is provided on Appendix A.

3.2.4 Job Scheduler (JS)

The Job Scheduler is the component that receives jobs when they are submitted to the framework by a Job Manager. Received jobs are added to a queue of pending jobs and are subsequently dispatched by the JS to an available PU according to a scheduling algorithm.

When a PU-M is started, an initial setup phase takes place: the OpenCL capabilities of all of its PUs, as well as performance metrics for each of them, are communicated to a JS. The JS then evaluates the quality of the connection between itself and the communicating PU-M. This is done by determining the latency and throughput between these components. These informations are subsequently stored as a set of properties associated with each PU. After this *registration* process is complete, the JS can then dispatch queued jobs to the PUs managed and made available by each of the registered PU-M.

The JS stores received jobs on its local job queue according to a certain order, determined by a given enqueueing policy. This can be a First-Come First-Served policy or, if desired, a policy that might reorder jobs according to some prioritization scheme. For example: jobs coming from a specific JM or jobs of a given category may be enqueued ahead of all others in order to antecipate the moment they start being processed on an available PU.

At the same time, the scheduler progressively dispatches enqueued jobs. According to a job dispatching strategy a job is chosen and, according to a workload allocation policy, an available PU is selected to process that job (Sections 2.5.2 and 3.3.4 describe typical scheduling algorithms and the ones implemented in the current prototype, respectively).

The decision on which PU must be selected for executing each job (the workload policy), as well as the dispatching strategy, can be configured by parametrizing the JS to the user's preference. Obviously, different algorithms render different scheduling decisions, and consequently different overall execution times for the same sets of jobs. The most adequate algorithms should be chosen for the environment on which the framework is being deployed. Given the differences between the multiprocessor architectures supported by OpenCL, running a code piece on a device may bring huge benefits in terms of program speedup, in comparison to running the same code on another device. On the other hand, if the system comprises PUs whose capabilities are all very similar, it might be irrelevant to opt for one or other of the available PUs. Various properties can be taken into account by a scheduling algorithm in order to perform better decisions such as the performance metrics retrieved at registration time pertaining to each PU, the link quality between the JS and each PU-M, the properties of each PU-M, and Jobs' task properties, attributes, as well as data sizes and types.

In order to acquire a more precise and updated view of the system, the JS receives informative data pertaining to the current state of each registered PU-M and corresponding PUs. This information can be actively queried by the JS or it may be sent by the PU-Ms, either after the occurrence of a given event or periodically, after certain amounts of time. After a job finishes being processed, a notification is sent to the JS by the PU-M, informing it of the occurrence. The JS can also receive system usage statistics such as PU usage rates and the number of jobs each PU has processed up to the moment. A periodic *heartbeat* can be sent from the PU-M to the JS so that it can keep track of PU-Ms that are accessible and those that may have disconnected; this heartbeat can be used to keep track and improve connection quality estimates as well. Job scheduling algorithms may use these informations in order to adapt their scheduling decisions to the current system in run-time. For example, some PUs may be overloaded by requests for processing many jobs, while others are being kept idle unnecessarily. Live system state informations can be used to detect and overcome such situations.

3.2.5 Processing Unit Manager (PU-M)

The Processing Unit Manager is the component responsible for receiving jobs dispatched by the Job Scheduler and launching them on the selected PU. When jobs complete, the output of their computations is sent by the PU-M to the Results Collector originally selected by the Job Manager.

As described on Section (3.2.4), when a PU-M is launched, a series of steps are taken in order to acquire a detailed view of the managed resources. A set of *static* properties are retrieved from the local environment such as available memory, processor types and clock frequencies. Additionally, the PU-M also carries out performance tests on each of the available OpenCL devices in order to determine device throughput (by means of a raw FLOPS measure) and the bandwidth of host-to-device memory transfers. During

the registration phase, the PU-M transmits these properties to the JS and answers to the connection quality inspection tests requested by the JS.

After registering with a JS, the PU-M is available to receive jobs and launch them on PUs for processing. A local job queue is maintained for each PU, where jobs are enqueued and dequeued following a FCFS order. Each job is processed only after the previous one for the same PU completed its execution. No rescheduling decisions are taken by the PU-M, as those were already made by the JS.

Each argument of a job is annotated by the JM as containing input, output or an auxiliary data space for use by the kernel. When preparing the execution of a job, arguments marked as *input* are initialized and their contents copied from the host to the OpenCL device memory. The other types of arguments only need to be allocated — no initialization is needed for them. After a job's execution completes successfully, the data on the arguments marked as *output* is returned to the requested RC. A new job on the same PU's job queue can be executed afterwards, repeating the aforementioned execution cycle. In case a job fails to be compiled by the OpenCL runtime compiler or a run-time error occurs, the data on its arguments is considered invalid. Such data is discarded and a failure message associated with the job is sent to the RC. The JM should process the received error codes appropriately.

Also after each job completes its execution, the PU-M sends a notification to the JS. The time taken to transfer the data in the job's arguments to the device and the kernel execution time are provided to the JS, to allow a more accurate view of the PU's performance and enable better scheduling options.

The PU-M also comprises a passive communications module for responding to status enquiries from the JS, as well as a periodic module for providing system status, allowing the JS to keep track of useful live system informations.

3.2.6 Results Collector (RC)

The Results Collector has two main functionalities: receiving data pertaining to job computational results sent by Processing Unit Managers and sending such data back to the Job Manager that submitted the job. This is the simplest component of the framework and its purpose is simply to behave as a buffer, holding results until they are claimed by the JM. This can be useful to alleviate the burden of storing potentially huge amounts of data that could accumulate and overload the PU-M or the JM.

Job results are received by the RC after they are sent from the PU-M. They are stored on an indexed structure, so that finding the results of a given job can be done quickly.

Jobs results can only be requested by the JM that created them. When a JM requests the results for a given job, and if such data is available, it is sent to the JM and deleted from the RC. If the data is not yet available, an error message is sent instead.

A JM may register a request on the RC to be notified when the results of a given job are available. This avoids frequent or at least periodic checks from the JM to know if

the desired data is available to be fetched. After receiving a notification request, the RC will send a notification to the corresponding JM as soon as the desired data is available. Afterwards, the JM can fetch such data.

3.3 Prototype

A prototype implementation was created to validate the proposed framework. Although initially intended to be a simplified version of the overall system, the current version evolved until the point where it includes most of the initially functionalities.

The current implementation was designed with extensibility in mind. The basic components of the framework can be considered modular enough so that future improvements can be made by modifying only specific parts of each component. Different local and distributed component deployment configurations are supported. Due to the nature of the components, deploying the framework under new system configurations should not represent a major change on their internal structure.

Various scheduling algorithms were implemented in order to test the framework. We designed a new, score-based workload allocation policy in order to provide a scheduling algorithm that is intrinsically designed from scratch to deal with heterogeneous computing units such as CPUs and GPUs under the same distributed system. The following sections further describe the most relevant details of the implemented prototype.

3.3.1 Dependencies on Third-Party Systems

The current version of the system is implemented in C. The communication between components is attained by a communications layer resorting to MPI [Mpia]. Currently there is currently a strict dependency on MPICH2 [Mpib], as this is the only stable implementation with support for applications with multiple (and possibly concurrently) communicating threads. Each job's task is expected to be a correct OpenCL [Mun10] source-code to be submitted to any OpenCL device. Using provider-specific extensions to the OpenCL specification on kernels submitted to the framework may result in undefined behaviour, depending on the device the kernel is processed on. All machines on which a component is launched must provide MPICH2 runtime support as well as necessary OpenCL libraries. PU-Ms require any needed GPU drivers installed and loaded in order to launch jobs onto GPUs.

3.3.2 Supported Configurations

The current implementation's components may be deployed in different configurations, being easy to adapt to different hardware setups.

The system may be deployed on a fully-local configuration, with one instance of each component running on the same machine. Alternatively, in order to take advantage of multiple hardware resources, the components may be scattered among various machines.

All configurations described on Section 3.2 may be deployed, except those that include deploying multiple levels of Job Schedulers, following a *meta-scheduler* approach. Although, deploying multiple, independent JSs is supported. A configuration involving a PU-M capable of managing a set of PUs on a group of disparate machines has not been tested.

3.3.3 Implementation Details

Some additional details about the implementation of the various components should be taken into account. We present the most relevant details of the domain-independent components:

Job Scheduler The JS, much as like as all of the other components, was designed with future expandability in mind. As a consequence, job scheduling algorithms may be implemented and replaced without considerable modifications to the core of the JS component. More specifically, all it takes at this point to implement a new job dispatching or workload allocation algorithm, is simply to create a new source file containing the necessary functions, edit one header file and to provide a string with the desired algorithm's name as an argument to the JobScheduler program executable.

The way the JS treats the data received by PU-Ms (both the static data retrieved at PU-M registration time, as well as live system information) depends on the workload allocation algorithm. The reception of this data is fully handled by the core JS, but how it is stored, managed and used by the scheduling algorithms depends on their implementation. Section 3.3.4 provides a detailed overview of each of the implemented algorithms and how they use the available PU informational data.

Processing Unit Manager The static properties of each device managed by the PU-M are collected at launch time from the OpenCL runtime installed on the machine. The type of each device, global and local memory sizes, clock frequencies, NDRange dimension limits supported by each device, as well as other properties, are all retrieved from the OpenCL runtime. This provides a platform-independent way of acquiring these details. Aside from these static properties, three additional performance tests are conducted by the PU-M:

1. Host-to-device memory throughput (GB/s) — a simple and intensive data transfer is performed between the host machine and the OpenCL device, using OpenCL buffer creation and writing functions. This allows to determine the peak data rate at which it is possible to transfer data from the host machine's RAM to each specific device's global memory;

2. Kernel compilation and submission latency (seconds) — this measures the time spent by *overhead* operations when submitting a kernel to a device. The time taken

for the creation, compilation, submission and retrieval of a simple kernel without any data transfers or processing taken on the device is measured;

3. Device throughput (seconds^{-1}) — a kernel that performs a significant amount of parallel processing is submitted to the device. The time taken to process this kernel is used to obtain a rough estimate on the processing speed of the device. The throughput is thus measured as the inverse of this kernel's execution time ($time^{-1}$: faster devices get higher values; slower devices get lower values). Currently, this is only an approximation to a more proper FLOPS measurement that should be implemented in the future.

OpenCL provides a queueing mechanism that allows for multiple kernels to be enqueued and that dispatches them onto the devices as they are available to process kernels. However, it was decided not to use this mechanism. Using the OpenCL queue mechanism would increase the complexity of synchronizing memory allocations onto the devices with the launching of the respective kernels. Huge memory allocations would have to wait until enough device memory was available, which only adds more complexity to the system. An external queue to OpenCL, in the PU-M's application layer is used instead.

All network transfers are processed independently of kernel processing. This means that while a kernel is being processed by an OpenCL device, the PU-M might receive more jobs in parallel from the JS. A relevant performance improvement is achieved by the fact that when a job finishes being processed, its output data is sent as soon as possible to the respective requested RC. A new thread is created for this purpose, allowing for the following job to be processed at the same time that network transfer takes place.

Results Collector Data received by the RC is currently only stored in RAM memory, by resorting to a binary search tree [Tse]. This was a decision made in order to improve insertion, search and deletion times with potentially huge amounts of data. The tree nodes are ordered by the unique identifier corresponding to each job-submitting JM, as well as the ID of each submitted job (result).

3.3.4 Available Scheduling Algorithms

All algorithms currently implemented follow a static scheduling approach and dispatch enqueued jobs as soon as possible. The JS does not wait for status updates or availability notifications from PU-Ms before dispatching any enqueued jobs. However, if a job is submitted to a JS, and a PU-M had already processed previous jobs and provided live system information, the JS may take that information into account.

3.3.4.1 Dispatching Strategies

All jobs received by the JS are enqueued by the order they arrive. Only one job dispatching strategy is implemented: a simple FCFS strategy. No prioritization scheme is used

and jobs are not reordered.

3.3.4.2 Workload Allocation Policies

Two main workload allocation policies were implemented: round-robin and a score-based workload allocation policy. The latter can be parametrized to customise its behaviour, and up to this moment two main variations were used: the *fixed* and the *adaptive* approaches.

Round-Robin Workload Allocation Policy (RR) The Round-Robin workload allocation policy selects PUs in a round-robin fashion. The absolute or relative performance properties of each PU are not taken into account by this policy.

Job attributes pertaining to PU type preferences are considered as follows. If a job is *required* to be processed on a certain PU type, only PUs of that type are considered for processing the job. Otherwise, if a job indicates *preferred* PU types, a PU of that type is chosen for processing if available. If no PU of type *preferred* is available, one of the available PUs with type identified as *allowed* is chosen. If no PU of type *preferred* or *allowed* is available, one of the available PUs of type marked as *avoid* will be chosen. Finally, if none of the previous succeeded, one of the types that is not marked as *forbidden* on the job is selected. The pseudo-code on Listing 3.1 shows this preference-based method used for selecting an eligible PU. The selectPU function simply selects one of the applicable PUs following a round-robin fashion, as shown on Listing 3.2. No live system information is taken into account by the RR allocation policy.

Score-Based Workload Allocation Policy The score-based workload allocation policy was developed in order to create an algorithm that better fits the balanced utilization of the heterogeneous computing resources targeted by the framework.

To select which of the PUs will execute a given job, the algorithm analyses the properties of the available PUs. Weighting the PU properties with the job's requirements and attributes, a score is given to each PU. The PU that returns the *best* score is selected for processing the job. All PU properties can be taken into account by the algorithm. Different parametrizations give different weight factors to each property, increasing or decreasing the relevance of certain aspects of the PU in spite of others. A weight factor of 0 (zero) can be used to ignore certain PU properties.

The implementation maintains a local registry for each PU containing the currently enqueued jobs, an approximation of the start time of the first job presently at the local queue and the number of jobs of each category that have already been processed. After a PU is selected for processing a given job, its local queue registry is updated with the information pertaining to this job.

When the JS receives a finished job notification from a PU-M, the queue registry for the corresponding PU is updated: the finished job is removed from the queue and the start time of the following job in the queue (if there is any) is updated. This information

```
bool allocateJobToPU (Job job) {
   bool success = false;

   if (job.requiresPUType)
     return selectPU(job, job.requiredPUType);

   if (job.hasPreferredTypes)
     success = selectPU(job, job.preferredPUTypes);

   if (!success && job.hasAllowedTypes)
     success = selectPU(job, job.allowedPUTypes);

   if (!success && job.hasAvoidableTypes)
     success = selectPU(job, job.allowedPUTypes);

   if (!success) {
     PUType_List allPUTypes = getAllSupportedTypes();
     allPUTypes.remove(job.forbidPUTypes);

     success = selectPU(job, allPUTypes);
   }

   return success;
}
```

Listing 3.1: PU selection by preference, pseudo-code.

```
PU_List allPUs; //contains all available PUs

bool selectPU (Job job, PUType_List possibleTypes) {
  PU_List suitablePUs = allPUs.getPUsOfType(possibleTypes);
  PU      selectedPU;

  suitablePUs.remove_PUs_without_enough_resouces_for(job);

  selectedPU = suitablePUs.head;

  if (selectedPU == NULL)
    return false;

  job.execOnPU = selectedPU;
  allPUs.remove(selectedPU);
  allPUs.tail  = selectedPU;

  return true;
}
```

Listing 3.2: Round-Robin PU selection, pseudo-code.

is later on taken into account by both the *fixed* and *adaptive* approaches. The number of jobs with the same category as that of the recently-finished job as well as the time that was spent for processing the job are also recorded on the registry. The *fixed* approach ignores these values, but the *adaptive* approach takes them into account.

The time spent for processing jobs of each category is stored following a weighted average method. If no jobs of the same category had been processed by that PU yet, the value that is kept is the same as what was reported by the PU. If any jobs of the same category had already been processed by that PU, the value that is kept is a time-weighted average between the stored value and the received new one. This has the effect of giving increasingly less weight to old values. Currently we use an empirical weight of 50% to the newly received one.

Given a dequeued job from the JS' queue, the general score-based workload allocation policy behaves as follows:

1. Job preferences on PU types are taken into account in the same way as under the RR allocation policy. Refer to Section 3.3.4.2 and Listing 3.1 for details.

2. From the list of PUs with suitable types, select those that have enough hardware resources and capabilities to process the given job (i.e., device global memory size, support for the requested NDRange dimensions, ...).

3. Determine the score for the given job on each PU. The PU that gets the best score is chosen for processing the job.

4. Update the local registry with information about the job.

The pseudo-code in Listing 3.3 shows the general structure of the score-based workload allocation policy.

***Fixed* approach** Under the fixed approach, live system information received by the JS pertaining to past job execution times on PUs is not taken into account. Parameters used by this approach are those acquired on the initial performance measurements made by PU-Ms on their local PUs, performance measurements over the communications link between the JS and PU-Ms, the number of jobs on the PUs' queues, and the start time for the first job on each queue.

The $scorePU$ function (Equation 3.1), as parametrized for the *fixed* approach, returns an approximation of the time at which a given job is predicted to finish on a given PU. This function adds the predicted time at which a job will be submitted by the respective PU-M to the PU, to the predicted time it will take to submit and process that job on the

```
PU_List allPUs; //contains all available PUs

bool selectPU (Job job, PUType_List possibleTypes) {
  PU_List suitablePUs = allPUs.getPUsOfType(possibleTypes);
  PU      selectedPU  = NULL;

  //Step 1
  suitablePUs.remove_PUs_without_enough_resouces_for(job);

  if (suitablePUs.length == 0)
    return false;

  //Step 2
  float bestScore = FLT_MAX;
  float currScore = FLT_MAX;

  forall(PU currPU : suitablePUs) {
    currScore = scorePU(currPU, job);
    if (currScore < bestScore) {
      bestScore  = currScore;
      selectedPU = currPU;
    }
  }
  if (selectedPU == NULL)
    return false;

  job.execOnPU = selectedPU;
  job.predictedFinishTime = bestScore;

  //Step 3
  selectedPU.addInfoToRegistry(job);

  return true;
}
```

Listing 3.3: Score-Based PU selection, pseudo-code.

PU.

$$scorePU(job, PU) = \begin{cases} startT(job, PU) + TTP(job, PU), \\ \qquad\qquad availableAt(PU) \leq startT(job, PU); \\ availableAt(PU) + TTP(job, PU), \\ \qquad\qquad availableAt(PU) > startT(job, PU). \end{cases} \tag{3.1}$$

The instant when a job starts to be processed can be predicted in one of two ways:

a) If the PU is predicted to finish processing all jobs on its queue *before* the given job is ready to start being processed, the instant at which the given job will be started is given by the $startT$ function (Equation 3.2).

$$startT(job, PU) = currTime + LinkLatency(PU) + TimeToTransfer(job, PU) \tag{3.2}$$

b) If the PU is predicted to finish processing all jobs on its queue *after* the given job is ready to start being processed, the instant at which the given job will be started is given by the $availableAt$ function (Equation 3.3).

$$availableAt(PU) = FirstJobStartedAt(PU) + \sum_{i=1}^{nEnqJobs} TTP(queJob_i, PU) \quad (3.3)$$

The time it will take for submitting and processing the job by the PU is given by the TTP function (Equation 3.4).

$$\begin{aligned} TTP(job, PU) = {} & JobSubmissionLatency(PU) \\ & + DeviceDataTransferTime(job, PU) \\ & + ProcessingTime(job, PU) \end{aligned} \quad (3.4)$$

The $startT$ function (Equation 3.2) returns the instant when the PU-M will dispatch a job to a PU, if the local queue of the PU is empty. Parameters used are the current system time ($currTime$), the latency of the connection between the JS and the PU-M ($LinkLatency$) and the time needed to transfer the job's data to the respective PU-M of the given PU ($TimeToTransfer$), obtained by multiplying the job's input data size (MB) by the inverse of the connection throughput [$(MB/s)^{-1}$].

The $availableAt$ function (Equation 3.3) returns a prediction of the instant when a PU will be next available. Parameters used by this function are the time at which the first job on the queue was started ($FirstJobStartedAt$), the number of jobs on the PU's local queue ($nEnqJobs$) and information maintained by the JS about the jobs on the queue, $queJob_1, \ldots, queJob_{nEqnJobs}$.

The TTP function (Equation 3.4) returns the predicted time to process a job on a PU. Parameters taken into account are the local job submission latency from the PU-M to the given PU ($JobSubmissionLatency$), the time to transfer the job's input data to the device ($DeviceDataTransferTime$), obtained by multiplying the job's input data size (MB) by the inverse of the host-to-device bandwidth [$(MB/s)^{-1}$] of the PU and the actual predicted time for processing the job on the PU.

The *fixed* approach ignores the processing times received by the JS after PUs finish processing jobs. The $ProcessingTime$ parameter is therefore constant for all jobs on each PU along the course of the JS's execution. The value used on this parameter a performance estimation measure for each device, as acquired at PU-M initialization time (currently, an approximation of FLOPS). The inverse of this value is used, representing the speed of each device (the lower the value, the faster the device). Like described earlier, the relevance of this parameter can be increased or decreased by resorting to a multiplication factor. Currently, this factor is fixed at 1 (one).

The consequence of this is that all jobs for each PU are seen as equal in terms of time

needed to be processed. Consequently, they end up being allocated to PUs proportionally on their processing capabilities, also taking into account the connections' properties and the time to prepare and initialize job execution on them.

***Adaptive* approach** Under the adaptive approach, the processing time of past jobs is taken into account. Job categories are assumed to be associated to jobs with similar characteristics and are used to compute an estimation of the job processing time. Parameters used by this approach are those acquired on the initial performance measurements made by PU-Ms on their local PUs, performance measurements over the communications link between the JS and PU-Ms, the number and categories of jobs on the PUs' queues, and the start time for the first job on each queue.

When a job of a certain category is dequeued from the JS' job queue, the algorithm uses the $AdaptScorePU$ function (Equation 3.5) to determine the score for each PU. This function behaves as follows:

a) If the given PU *has not* been selected for processing a job of the same category as that of the current job, the returned score is the lowest possible: 0 (zero);

b) If the given PU *has* already been selected for processing a job of the same category as that of the current job, the returned score corresponds to the value returned by the $scorePU$ function (Equation 3.1).

$$AdaptScorePU(job, PU) = \begin{cases} 0 & , \quad \neg alreadySubmitted(job, PU); \\ scorePU(job, PU), & alreadySubmitted(job, PU). \end{cases} \tag{3.5}$$

The $alreadySubmitted$ function returns $true$ if a job of the given category has already been submitted to the given PU. Otherwise, it returns $false$.

Observed processing times are updated under this approach, which means that the $ProcessingTime$ value of the TTP function (Equation 3.4) will correspond to a value that was communicated from the PU to the JS. However, when the PU has not yet finished processing its first job of the same category, such value has not been acquired yet. In that case, the $ProcessingTime$ value will correspond to the same performance indicator as what is used under the fixed approach (the initial estimation of the processor speed).

The overall behaviour of the adaptive approach can be described as follows:

1. Jobs of a given category $\mathcal{C}$, are initially submitted to each PU in a RR fashion;

2. After all PUs have been submitted at least one job of category $\mathcal{C}$, subsequent jobs of the same category are submitted following an allocation policy similar to that of the fixed approach;

3. When PUs start providing the processing times of jobs of category $\mathcal{C}$, these values are taken into account, and the observed processing times are considered when dispatching future jobs to those PUs.

3.4 Summary

This chapter presented a framework that aims at improving the utilization of all computing resources in heterogeneous computer systems, both on fully local systems and on multiple distributed machines. The chapter started with a description of the design of the overall system, followed by a detailed description of all of the components that comprise the framework, namely: the Job Manager, the Job Scheduler, the Processing Unit Manager and the Results Collector. The structure of a job was also described, as well as an API that can ease the development process of programs that use the framework. Subsequently, we provided some details of the prototype that we developed, and that allowed to validate and further assess the capabilities of the framework and its functionalities. Two workload allocation policies were implemented: round-robin and score-based. Their behaviour, in the context of the framework's Job Manager, was described in detail.

In the following chapter, the presented framework is evaluated and benchmarked, and the performance of the different algorithms is assessed under different utilization scenarios.

4

Evaluation

4.1 Introduction

In this chapter we present the experimental work performed to assess the validity as well as the performance gains our framework brings forth.

In Section 4.2 we present the hardware that was used for each of the performed experimental tests. Section 4.3 describes the criteria used to evaluate the results of each of those experiments. In Section 4.4, the two applications that were used to validate the framework are introduced and, afterwards, the benchmarking results and corresponding achieved speedups are presented and discussed.

4.2 Experimental Settings

For each application, three sets of tests were performed in order to evaluate the performance of their original implementations, their behaviour under an OpenCL environment and the performance gains achievable by resorting to the present framework.

- The first set of tests consisted on an assessment of the execution time of the original, sequential versions of the existing algorithms. These tests were performed under a strictly local environment, using a single CPU core of each of the available machines. The list and characteristics the machines that were used is depicted in Table 4.1.

- The second set of tests aimed at evaluating the behaviour of the different available PUs under different OpenCL kernel configurations. These tests used the implemented framework and were conducted under a local configuration. A single

instance of each component of the framework was deployed, and all components were running on the same machine.

- Tests in the third set aimed at reducing the execution time of the original programs. Under this setting, the Job Manager, the Job Scheduler and the Results Collector were all launched on the same machine. Multiple Processing Unit Managers were deployed, one on each of the remaining machines. These machines hosted both multiprocessor CPUs as well as consumer-grade and scientific computing-targeted GPUs (see Table 4.1). An initial subset of benchmarks restricted the usage of PUs to only CPUs. The subsequent set of benchmarks allowed both CPUs and GPUs as eligible processors for the submitted jobs. The job configuration and attributes used on the distributed tests were inferred from an analysis of the behaviour displayed by the different PUs on the previous tests.

Table 4.1: Hardware details (horizontal rules separate PUs on different hosts).

Designation	PU type	Comp. Units[1]	Clock	RAM
SunFire X4600 M2	CPU	16	1.00 GHz	32 GB
Intel Core 2 6420	CPU	2	2.13 GHz	2 GB
NVIDIA Quadro FX 3800	GPU	24	1.20 GHz	1 GB
Intel Xeon E5506	CPU	4	2.13 GHz	12 GB
NVIDIA Quadro FX 3800	GPU	24	1.20 GHz	1 GB
NVIDIA Tesla C1060	GPU	30	1.30 GHz	4 GB
NVIDIA Tesla C1060	GPU	30	1.30 GHz	4 GB
Intel Core i5 650	CPU	4 ($2 \times$ 2HT)[2]	3.20 GHz	4 GB
NVIDIA GeForce GTX 480	GPU	15	1.40 GHz	1.5 GB
Intel Xeon 5150	CPU	4 ($2 \times$ 2HT)[2]	2.66 GHz	2 GB

[1]OpenCL-reported Compute Units. For CPUs, this is equivalent to the number of virtual CPU cores.
[2]Processors with HyperThreading technology.

For the distributed tests, the JM, JS and RC were launched on an additional computer with an Intel Core 2 Duo T6400 processor (totaling 2 cores) at 2.0GHz and 4 GB of RAM. Communication between different machines was performed over a virtual private network overlaying a 100 Mbit LAN.

All tests were conducted with MPICH2 version 2.1.1 [Mpib]. OpenCL runtime for CPU execution was supported by the OpenCL libraries provided by ATI's Stream SDK, version 2.2 [Atib]. All GPUs that were used are NVIDIAs, so NVIDIA's OpenCL libraries as well as NVIDIA graphics driver for Linux, version 260.19.14 [Nvi] were used for supporting kernel execution on GPUs.

The results of the benchmarking tests executed when launching the framework are depicted in Figure 4.1. Chart 4.1a shows the time needed for the creation, compilation and submission of a simple kernel on each PU. Although there is a noticeable difference

on kernel submission latency for the SunFire one should note that the time unit under consideration is milliseconds. This means that these absolute values end up being of no impact in the execution time of longer-lasting jobs, that need more than a couple of seconds to be processed. Chart 4.1b shows the bandwidth of the bus used for data transfer from the respective PU-M's main memory to the PU memory. This might be a transfer from the computer's main RAM to a GPU's dedicated memory space or to a memory space visible by the OpenCL runtime for CPU kernel execution. It is interesting to note that the average data copy bandwidth to the available GPUs outperforms the average RAM-to-RAM performance for the available CPUs. In Chart 4.1c, we show the absolute, parallel performance of each PU as determined by the execution of a parallel, benchmarking job (somewhat comparable to a FLOPS measurement). The parallel processing capabilities of GPUs are quite noticeable, with the GTX 480 evidencing itself as a very, very fast parallel processor. The relative speeds between CPUs are also well reflected by the charted values. Higher-resolution versions of these charts can be found in Appendix B.

It must be noted that the initial benchmarking tests are portraying an imprecise view of the computational properties of the PUs. Although processor intensive, the initial test job is a simple SIMD computation with a small memory footprint. A job of this kind may be subject to caching and other optimizations on some PUs and not on others. When more complex and longer-lasting jobs are submitted to these PUs, their relative performances may differ. Ideally, these initial benchmarks should be regarded as performance *indicators* and not as definite assessments on the available processors.

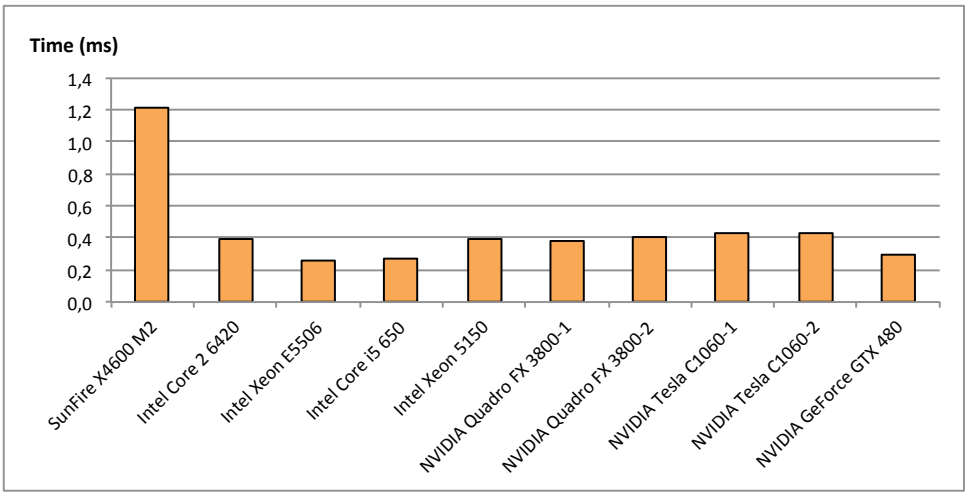
4.3 Evaluation Criteria

The framework was evaluated according to three main criteria:

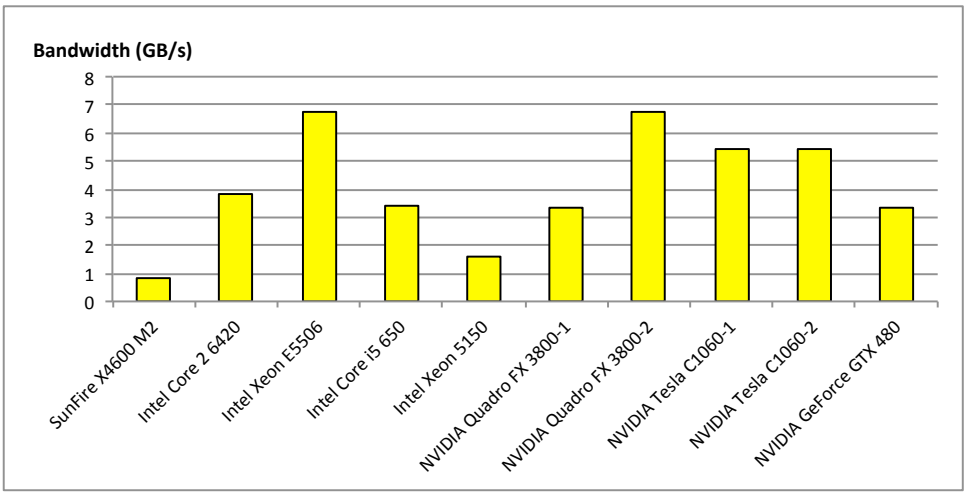
1. The impact of the introduction of a new type of processing units (GPUs) in an environment where only CPUs were used — A first subset of tests restricts the usage of PUs to only CPUs. For the subsequent set of tests, both CPUs and GPUs are allowed. Afterwards, we compare the turnaround times observed by the Job Manager for both cases.

2. The effect of submitting different numbers of jobs to solve the same problem — Various tests are performed in which different numbers of jobs are created. Creating less jobs implies more computational requirements on each job; creating more jobs implies fewer computational requirements on each job. The impact of such work divisions on the overall turnaround times is evaluated.

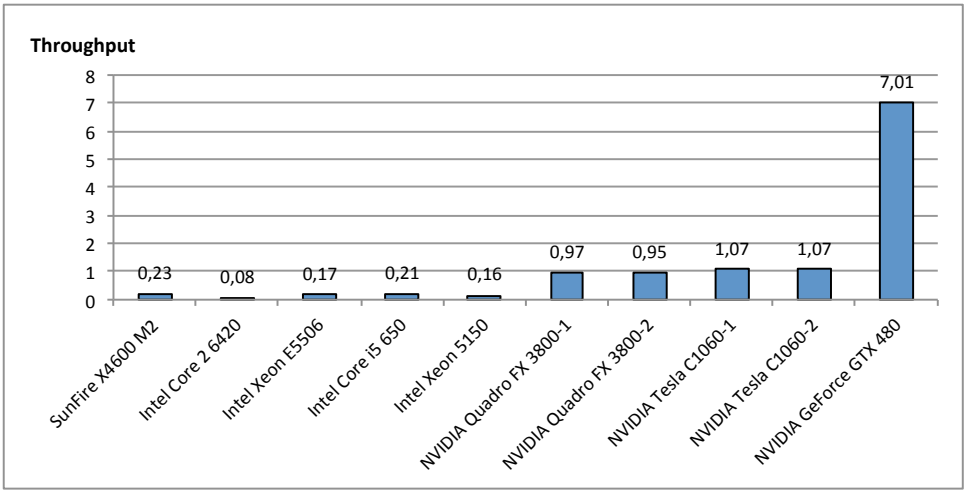
3. The achieved speedups over the original versions of the algorithms — The execution times of the original, sequential versions of the algorithms are compared to those achieved when resorting to the framework under a distributed setting.



(a) Kernel Submission Latency



(b) Data Transfer Bandwidth



(c) PU Throughput

Figure 4.1: PU properties inferred by the Processing Unit Managers

The overall turnaround times observed by the Job Manager are measured since the first job is submitted until all of the corresponding results are fetched.

4.4 Testing Applications

Two main applications were developed to evaluate the performance gains brought forth by the framework. The first is a Mandelbrot Set renderer; the second is Fdist, a gene identification algorithm, used for identifying differentiated genes on a population.

Three sets of tests were performed: the first consisted on the execution of the original, sequential algorithms. The second was a set of local tests conducted in order to investigate and better understand the behaviour of the different available PUs under different OpenCL configurations. Finally, a series of distributed tests were performed in which different scheduling algorithms were tested, under both CPU-only as well as CPU+GPU environments. The job configuration and attributes used on the distributed tests were inferred from an analysis of the behaviour displayed by the different PUs on the previous tests.

4.4.1 Mandelbrot Set Renderer

Mandelbrot Set [Man83] generation programs are well-known applications in which an image can be computed with a theoretically infinite level of detail. The Mandelbrot Set is analogous to many scientific observations in which, depending on the distance of the observer to the subject, more or less detail can be detected. This test application renders a highly detailed area of a Mandelbrot Set. Aside from being a very processor-intensive algorithm, it can also be considered an embarrassingly parallel problem. Each pixel of the obtained canvas can be computed independently of all others, and there is absolutely no need for synchronization or communication between the different processors executing the algorithm.

A sequential C program was used as the original implementation of the algorithm. The amount of processing for this algorithm is parametrized by the number of possible colours that are expected in the final rendered image. This is directly reflected on the number of possible iterations of the most computationally-intensive cycle of the program. Regions with mostly black pixels take longer times to be computed, as the determination of a black pixel yields the longest-lasting computations; coloured regions are overall faster to compute. The amount of data to be processed is determined directly by the width and height, in pixels, intended for the final image. Despite these differences in processing requirements between pixels, when adapted to be run in parallel, this application maps well to a SIMD execution model.

An OpenCL kernel was directly derived from the original implementation, each item computing an horizontal strip of the final image. For example, for an image of 1024×1024 pixels, if 512 items are to process it, each one computes a strip of 1024×2 pixels. When

generating multiple jobs, the canvas is divided in equal parts for each job, also in horizontal strips.

4.4.1.1 Experimental Results

Sequential Benchmarks The chart in Figure 4.2 shows the execution times of the Mandelbrot Set renderer under sequential execution (using a single CPU core). Execution times are shown for both the original, C version (in blue/dark), and for the OpenCL implementation (in yellow/light) executing with a single item, therefore, in a single processor. The program was parametrized to render a 10240×10240 pixel canvas, with 1,048,576 possible colours. This configuration generates a raw image with 300 MB.

Figure 4.2: Execution times for the original Mandelbrot Set rendering program on the available CPUs.

It took several hours for the algorithm to complete in all machines. The Intel Core i5 CPU displayed the fastest execution times, averaging at 5h 2min 20s. The Intel Core 2 6420 CPU showed the worst execution times, with 10h 9min 42s. There is a visible overhead introduced by OpenCL on most machines. For the processors where this overhead is not evident, the execution times are very similar to those of the original C implementation. On average, the OpenCL version introduced an overhead of 4.32%.

Local, Configuration Evaluation Tests The chart in Figure 4.3 shows the performance behaviour of a single job, parametrized to render a 1024×1024 pixel canvas, with 1,048,576 possible colours (rendering a 3 MB image). The job was submitted under a framework setup using only a single node with the number of global items was fixed 512 and along a single dimension — one-dimensional NDRange[1]. The tests benchmarked the performance of the devices with different group size configurations.

[1]Most PUs have a limit of 512 items per group, so a decision was made to only test configurations that could be acceptable for all available PUs

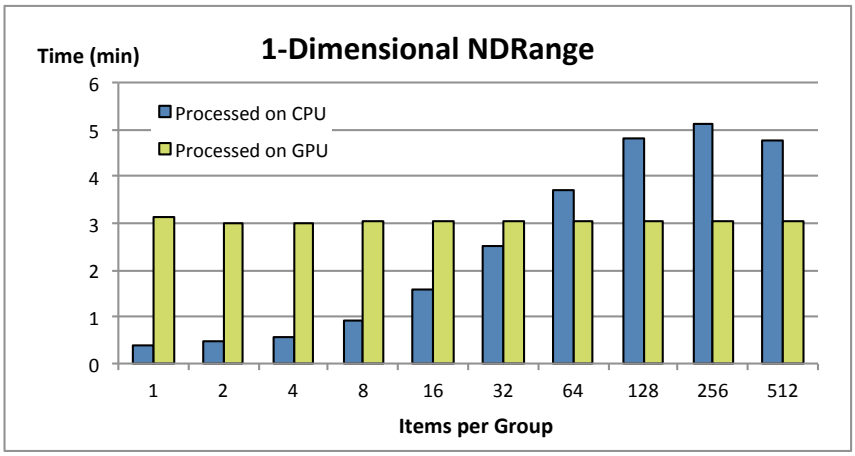


Figure 4.3: Performance behaviour of the Mandelbrot Set kernel with a 1-dimensional NDRange. Comparing CPU (dark/blue) and GPU (light/green) performances.

Blue/dark bars show the execution times under a CPU. Green/light bars show the execution times under a GPU. As all PUs of the same type displayed a similar behaviour, we only present the absolute numbers for one PU of each kind: the SunFire X4600 M2 for CPUs and the NVIDIA GeForce GTX 480 for GPUs.

When running on CPUs, it is clear that the algorithm's performance worsens when group sizes are larger. When groups are defined as containing a single item, we observe the best performance values. This is expected, as there should be no benefit from grouping SIMD items that present no synchronization requirements. On the other hand, when using groups of 512 items, a slight decrease in the execution time is perceivable. A possible explanation, which requires further testing for verification, might be that with a group of size 512 items and a total of 512 global items, only a single group will be present. This might reduce to a certain extent the overhead of group management by the underlying OpenCL runtime.

As for the performance observed when running on GPUs, a different behaviour was observed: job run time was practically constant independently of the work-group configuration.

The chart in Figure 4.4 depicts the performance behaviour of a job launched under similar conditions as those of the previous tests (Figure 4.3), but using a two-dimensional NDRange, totalling 512×512 items. The values charted on the horizontal axis correspond to the number of items per group dimension — the total number of items per group is determined by raising these values to the second power. The total items per group is limited by both AMD (which supported CPU execution) and NVIDIA's OpenCL implementations to a maximum of 1024 items per group, and for that reason no more than 32 items per group dimension can be used.

The variation in the behaviour of CPUs is comparable to that observable under the one-dimensional NDRange tests. On the other hand, when observing the algorithm's

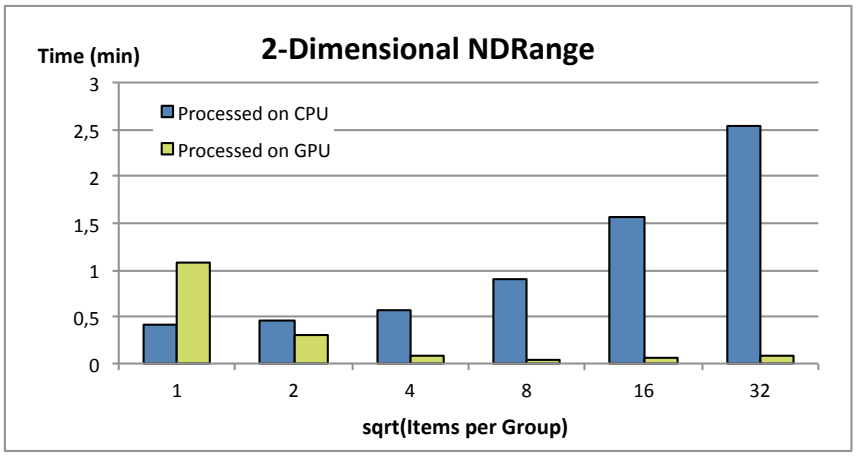


Figure 4.4: Performance behaviour of the OpenCL Mandelbrot Set kernel, 2-dimensional NDRange, CPU and GPU compared.

performance on the GPU, there is a very meaningful change. By resorting to a two-dimensional NDRange, with a single item per group, job runtime was already at about one third of the time observed under the one-dimensional configuration (Figure 4.3). This might be explained by the fact that now, many more global items are processing the algorithm in parallel, thus reducing overall processing time. When grouping items into larger groups, GPU execution performance kept improving. This behaviour was unexpected due to the SIMD nature of the application, but it might possibly be explained by the underlying characteristics of GPU hardware, where groups of processors are associated and designed to run in a concerted SIMD manner (see Section 2.4.1).

Distributed Benchmarks The chart in Figure 4.5 depicts the turnaround times observed by the Job Manager for different numbers of jobs under a distributed environment. Submitted jobs were configured to use a two-dimensional NDRange of 512×512 items and groups of 8×8 items. The algorithm is parametrized for rendering 1,048,576 possible colours and to generate a canvas of 10240×10240 pixels. The Job Scheduler is configured to resort to the fixed score-based workload allocation policy.

Blue/dark bars indicate turnaround times when all submitted jobs were defined as required to be processed on CPUs, while green/light bars correspond to the observed turnaround times when jobs were allowed to execute in both CPUs and GPUs.

When using only CPUs, it is noticeable that more jobs provided better turnaround times. A clear performance drop is visible when less than or as many as 8 jobs were submitted, when compared to the submission of 16 or more jobs. By analysing the Job Scheduler's execution logs, it was possible to understand that, for just a few jobs, because under the fixed approach the first jobs arriving at the scheduler are dispatched to the faster machines, the jobs that arrive later are dispatched to slightly slower machines. In the case of the 8 jobs, for example, this results in the last two jobs being submitted to the two Xeon machines. Coincidentally, these two last jobs correspond to the lower region

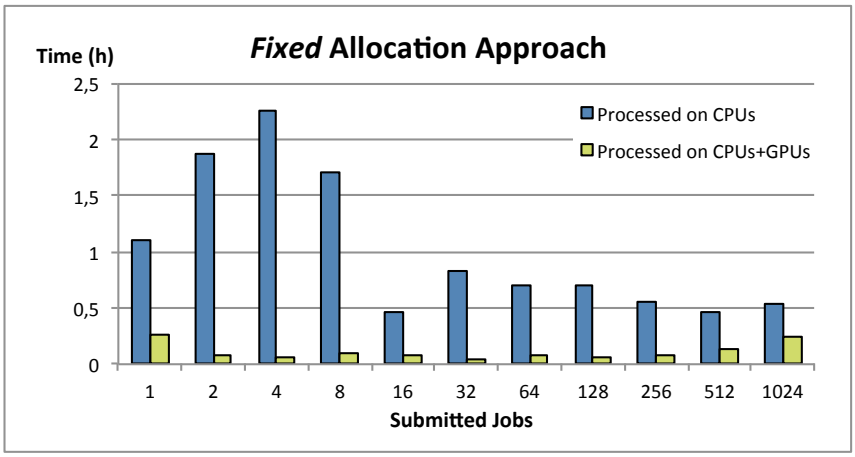


Figure 4.5: Turnaround times observed by the Job Manager for the distributed Mandelbrot Set computation using the fixed workload allocation approach. CPU-only and CPU+GPU compared.

of the Mandelbrot Set detail being computed, and are in fact the most computationally-intensive regions, resulting in these machines being used for the heaviest computations. With 16 and more jobs, this region was partitioned in much smaller chunks, and its computational impact was less noticeable.

Including GPUs as eligible processors had a very noticeable positive impact on the overall turnaround time. The longest turnaround time was of 15 minutes, when only a single job was submitted, while the shortest was of a mere 2 minutes and 35 seconds, with 32 jobs. Refer to Figures B.3a and B.3b in Appendix B for a more detailed view of these charts.

The turnaround times for the adaptive approach are depicted in Figure 4.6. All job attributes were configured to the same values as those used under the fixed approach. Additionally, all jobs were attributed the same category.

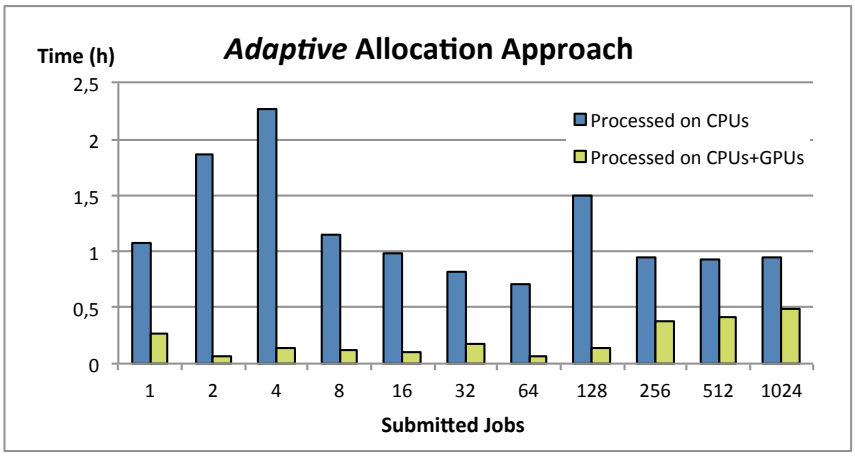


Figure 4.6: Turnaround times observed by the Job Manager for the distributed Mandelbrot Set computation using the adaptive workload allocation approach. CPU-only and CPU+GPU compared.

Before initiating all benchmarking sessions for this approach, a Mandelbrot Set rendering job calculating a reduced canvas of 1024×1024 was sent to each PU. These *profiling* jobs had the same category as the following benchmark's jobs, so that their execution time was recorded by the JS and taken into account for the subsequent score attributions (see Section 3.3.4.2).

Overall turnaround times were, on average, worse under this approach than those observed when resorting to the fixed approach. We believe that this might have happened because the initial profiling jobs, being representatives of the *average* computation for the whole Mandelbrot Set detail, were not representative of the *majority* of the subsequent jobs. This ended up misleading the adaptive algorithm and thus provided worse scheduling decisions than when merely taking into account the raw parallel processing speed of the PUs.

Once more, including GPUs provided much lower turnaround times, in all cases still better than under the fixed approach when using only CPUs.

Achieved Speedups The best turnaround times were achieved under the fixed allocation approach by submitting 32 jobs to both CPUs and GPUs. The corresponding overall turnaround time, observed by the JM, was of 2min 5s, a speedup of 116.7 × to the fastest (on the Intel Core i5 CPU) and 235.3 × to the slowest (on the Intel Core 2 CPU) sequential execution times achieved with the original algorithm.

Under the adaptive allocation approach, best results were also obtained when combining CPU and GPU processing power, with 4min 23s when submitting 64 jobs. This represents a speedup of 68.8 × to the fastest and 138.8 × to the slowest execution times achieved with the sequential, original algorithm.

4.4.2 Identification of Genes Potentially Under Natural Selection

Genes that differ substantially relatively to the gene pool where they belong tend to be subject to natural selection. The Fdist application [BN96] simulates a coalescent process [Hud91] generating many simulated neutral genes under an island model [Wri43]. Heterozygosity and Fst [WC84] are calculated for all simulated loci. Empirical datasets are compared to simulations with the same average Fst. All genes that are outliers to the surface made with Heterozygosity and Fst are deemed to be candidates for being under selection. This can be used, for example, to determine what specific genes are associated with skin and hair colour, what genes are susceptible to certain viruses such as the HIV, genes that are resistant to treatments, genes that influence the amount of meat available on livestock animals, as well as many other differentiating indicators.

The results of a simulation are used to render a chart indicating the probability of occurrence for specific markers (i.e., the Heterozygosity/Fst surface described above). Genes that fall outside the charted area to a certain confidence interval (typically 95% or 99%) are considered to have been potentially influenced by natural selection. By defining

the number of simulated genes to very high values (millions), a smother surface can be generated than one typically achieved by simulating thousands of genes.

By the nature of this application, the simulation of each gene is independent from all others and no synchronization or communication is necessary between the simulation of different genes. However, due to a random behaviour inserted onto certain parts of the algorithm, each run may diverge significantly from the others. On the developed OpenCL kernel, each item simulates an equal subset of the requested number of total genes. Items may diverge significantly on their execution state, meaning that this application maps well to a MIMD execution model, and not a SIMD execution model. When generating multiple jobs, each one simulates an equal subset of the overall number of intended genes.

4.4.2.1 Experimental Results

Sequential Benchmarks The original, C version of the algorithm was executed on the available CPUs. Being a sequential implementation, only a single core of each CPU is used. The algorithm was configured to simulate 5,120,000 genes, and generates a 90 MB output file. The OpenCL version of the algorithm was also processed on each CPU in a stand-alone configuration (without attachment to the framework) and with a single-item configuration, i.e., sequentially.

The original implementation of the algorithm takes a few hours to be processed on all CPUs. The OpenCL version displays consistently worse execution times. The Core i5 CPU provided the best execution times, with 3h 56min 50s. The Core 2 was the slowest processor for this program, executing it in 7h 21min 51s. The chart in Figure 4.7 shows the observed execution times both with the original, C implementation (dark/blue bars) and with the OpenCL implementation (yellow/light bars).

Figure 4.7: Execution times for the original gene simulation program on the available CPUs.

Local, Configuration Evaluation Tests The Chart in Figure 4.8 depicts the performance behaviour of a single job under a local, single-PU environment. The job was parametrized for using 512 global items and for simulating 5120 genes, generating a 90 KB output file. The framework was configured in a similar way as that for the local tests of the Mandelbrot Set algorithm (only one computer, all components local). Similarly to what was observed under the Mandelbrot tests, all PUs of the same type displayed a similar behaviour, and for that reason we only show the absolute values obtained with the SunFire CPU and the NVIDIA GTX 480 GPU. Blue/dark bars show CPU execution times and green/light bars show the execution times under the GPU.

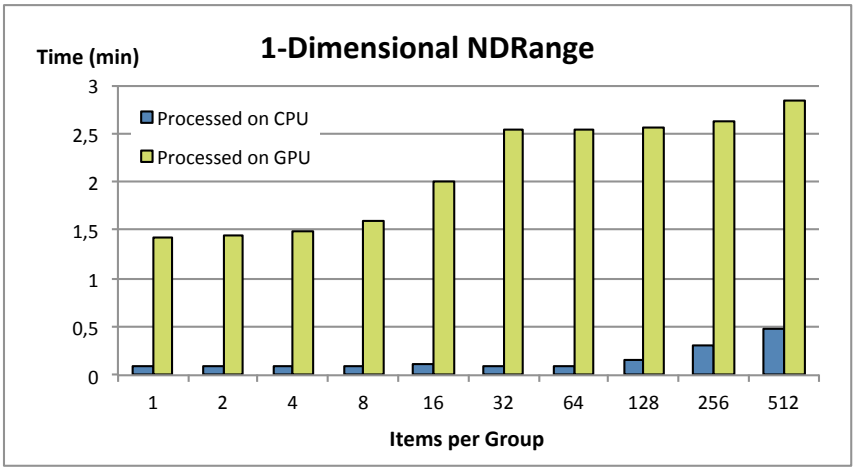


Figure 4.8: Performance behaviour of the OpenCL gene identification kernel, 1-dimensional NDRange, CPU and GPU compared.

In this case, there is an evident performance decrease when the number of items per group is increased, both with CPUs as well as with GPUs. It is also visible that GPU performance is consistently much worse than what is attained with a CPU. These results should be expected as this is a MIMD, complex application, and GPU hardware is currently not optimized for these types of programs.

The OpenCL version of this algorithm has huge memory requirements, limiting the problem to at most 512 items per job in the available hardware. Consequently, for the two-dimensional tests, jobs were configured to use a number of items that was 484, the nearest square number to 512 — meaning that the global work-item configuration was set to 22×22 items. The number of simulated genes was defined to 4840, thus making each item simulate exactly 10 genes. Various numbers of items per group were tested and the resulting PU behaviour is displayed in Figure 4.9.

The obtained performance values were very similar to those achieved under similar one-dimensional configurations (groups are *similar* when their have a similar number of items under a two-dimensional and a one-dimensional NDRange: a 2^2-item, 2D group is similar to a 4-item, 1D group; an 11^2-item group is similar to a 128-item group and a 22^2-item group is similar to a 512-item group). When compared with the one-dimensional configuration, meaningful performance improvements were not observed when using

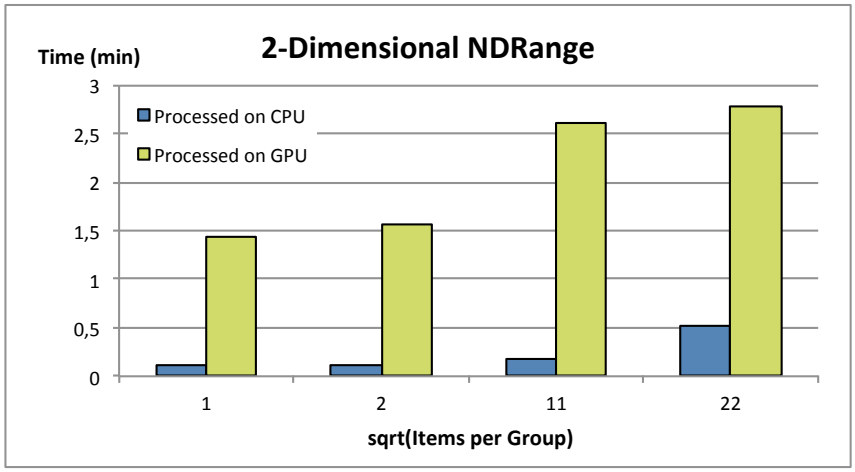


Figure 4.9: Performance behaviour of the OpenCL gene identification kernel, 2-dimensional NDRange, CPU and GPU compared.

both CPUs or GPUs with a two-dimensional configuration.

Distributed Benchmarks For the distributed benchmarks, jobs were submitted with a one-dimensional NDRange configuration. each job was configured to use 512 items and a single item per group. An output file of 90 MB is generated from the simulation of a total of 5,120,000 genes on each test. Different numbers of submitted jobs were tested for each configuration of the JS's score-based workload allocation policy. On these charts, blue/dark bars show the turnaround times observed when all submitted jobs were configured to only be dispatched on the available CPUs. Green/light bars show turnaround times when the submitted jobs allowed being processed on both CPUs and GPUs.

The Chart in Figure 4.10 pertains to the obtained turnaround times when the Job Scheduler was configured to use the score-based workload allocation policy, with the fixed configuration.

The first, obvious, and very noticeable observation is that the overall observed turnaround time when GPUs are allowed to be used is much worse than what is observed when only CPUs are allowed. Performance when using CPUs can hardly be seen under the scale of this chart. A more detailed view of the turnaround times for CPUs is available in Figure B.5a, in Appendix B. For a CPU-only environment, performance is best when 5 jobs are used, and there is a visible peak on the observed turnaround time when 2 jobs are used. When a single job was submitted, it was processed by the SunFire PU. With two jobs, the first was sent to the SunFire and the second to the Core i5 PU. It was observed that the latter processor displayed speedups of only 2× when running this kernel in parallel, in spite of having 4 virtual CPU cores. Taking this into account, two main factors might have lead to the observed turnaround time for two jobs: first, the fixed approach takes into account the raw, parallel processing speed of the devices, independently of the jobs' processing characteristics; second, the observed speedup (2×) for this processor might be due to the fact that this is an HyperThreaded CPU, which might not be able

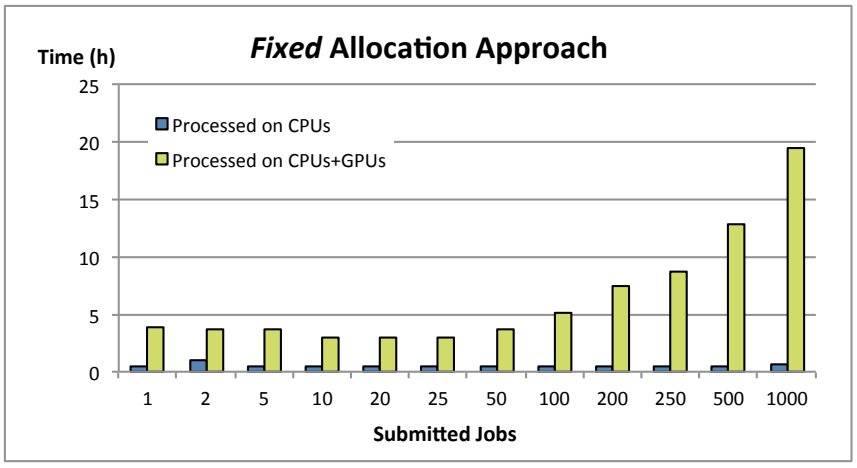


Figure 4.10: Turnaround times observed by the Job Manager for the distributed gene identification program using the fixed workload allocation approach. CPU-only and CPU+GPU compared.

to fully parallelize this MIMD application on each physical core. The JS' recorded raw speed for this PU indicated much higher speeds for the processor than those attained for this specific application, thus providing this sub-optimal scheduling decision.

When using GPUs, submitting between 10 and 50 jobs gives the best results. Performance decreases when using 100 and more jobs. This is explained by the fact that the Tesla and Quadro FX GPUs were much slower for this problem than the other available PUs (about 10× slower than the GTX 480, about 100× slower than the CPUs). With the fixed approach, and because raw PU speeds are taken into account, more jobs are being sent to these machines, worsening the overall processing time. When fewer jobs are submitted, a smaller impact is observed on the overall turnaround time because these GPUs receive less jobs.

Finally, the Job Scheduler was configured to use the adaptive workload approach. The resulting turnaround times observed by the Job Manager are seen on the Chart in Figure 4.11. Jobs were submitted with a configuration similar to that used with the fixed approach benchmarks. The initial, *profiling* jobs generated 5120 genes and can be said to be relatively accurate representatives of the forthcoming jobs' computations.

The achieved results show that good turnaround times were obtained, both when using only CPUs and when using CPUs and GPUs. It is visible that using only a few (less than about 5) or too many (more than 200) jobs renders worse results. This is understandable because when too few jobs are submitted, some PUs will be idle while others will be used intensively. When using many jobs, each job computes a very small portion of the overall intended computation, and the data transfers and overheads introduced by each component become more prevalent, increasing the overall turnaround time observed by the JM.

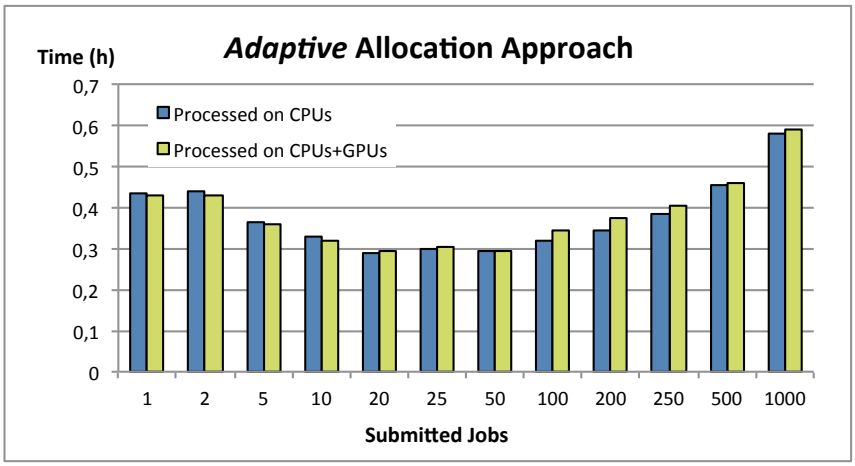


Figure 4.11: Turnaround times observed by the Job Manager for the distributed gene identification program using the adaptive workload allocation approach. CPU-only and CPU+GPU compared.

Achieved Speedups The best results for the gene identification algorithm with the fixed approach were attained when using 5 jobs and by allowing only CPUs to be used. The JM observed a total turnaround time of 23min 29s. This is a speedup of 10.1× relatively to the fastest and 18.8× to the slowest sequential executions, observed on the Core i5 and the Core 2 PU, respectively.

With the adaptive allocation approach, best times were achieved when submitting 20 jobs that only allowed execution on CPUs. A turnaround time of 17m 13s was observed. This is a speedup of 13.7× to the fastest and 25.6× to the slowest sequential executions.

4.5 Summary

In this chapter we have shown the experimental work performed to validate and assess the capabilities of the proposed framework. We have also described the hardware setting where the validation benchmarks were executed.

We presented results for two distinct types of applications. First, we tested and benchmarked a Mandelbrot Set renderer — a SIMD application. Afterwards, we tested and benchmarked an application that implements a gene identification algorithm, used in a real-world scientific computing setting that follows a MIMD approach.

The original sequential versions of both applications had huge execution times (several hours in any of the available CPUs). The applications were adapted to use the proposed framework and benchmarked with the scheduling algorithms proposed in the Chapter 3. Very meaningful results were achieved for the SIMD application, with speedups of up to 253.3×, where the addition of GPUs clearly benefited the algorithm's overall performance. For the MIMD application, it was observed that GPUs did not perform well, but the proposed score-based adaptive workload allocation policy, provided very acceptable and improved overall execution times. The performance of the algorithm was not

impaired by the usage of sub-optimal processors, that in some instances even contributed to reducing the observed turnaround times. Speedups of up to 25.6× were achieved for this application.

The next chapter presents an overview of the work described on this dissertation and provides some possibilities for future work.

5

Conclusions

5.1 Conclusions

HPC applications have huge computational requirements. In order to process a problem with high processing demands, research companies and organizations can either resort to dedicated computer clusters or rent some computing time in so-called *cloud services*. However, these solutions usually require software and hardware specialists for configuration and maintenance of complex infrastructures or are too expensive for the organization in question. With the emergence of GPGPU, inexpensive, high-performance processors are available at very reduced costs for these organizations.

The present dissertation proposed a framework and its corresponding implementation that takes advantage of existing computing infrastructures where a number of multiprocessor CPUs, as well as GPUs with General-Purpose computing capabilities, are readily available. This framework targets at greatly accelerating HPC programs and scientific computing applications. It is composed of a set of components that are sufficiently flexible so that they may be deployed under different system configurations to better fit the hardware infrastructure. Various configurations such as single-computer multi-PU machines, Clusters of computers or even multi-site COWs are supported. The component of the framework that is responsible for job scheduling is configurable and allows for the usage of different scheduling algorithms. For this end, a set of scheduling algorithms were proposed in order to better take advantage of the architectures available under such heterogeneous environments.

The proposed framework was validated by implementing a complete and fully functional prototype, on which two applications of a different nature were tested. The first is a SIMD synthetic application that generates a detailed region of a Mandelbrot Set. The

second is a MIMD program that generates genes for a real-world scientific computing application. The total accumulated time of the tests is on the order of thousands of hours, which makes us believe that the current implementation of the framework can be considered stable enough to be used by scientific computing domain specialists not accustomed to parallel or distributed computing.

Very significant performance improvements were achieved by using the framework for both applications. Given an appropriate scheduling algorithm is selected, the framework is capable of selectively choosing appropriate devices, depending on the requirements of the submitted jobs. The historical, observed behaviour of previously submitted jobs can be taken into account, which allows to optimise subsequent scheduling decisions. Under a distributed environment, and resorting to five CPUs and five GPUs, we achieved speedups of one to two orders of magnitude, with reduced effort on the development of the specific applications.

5.2 Future Work

Further research is needed to better evaluate which job scheduling algorithms better combine the problem characteristics with the system properties to maximize job throughput. This includes the development of more sophisticated job scheduling algorithms, tailored specifically for job allocation under heterogeneous systems that include multicore CPUs and GPUs. All job scheduling algorithms available in our prototype follow a *static* approach, in that jobs are dispatched from the JS as soon as possible, and may stay on the local queues of the PUs for a long time. This could be optimized by letting some jobs wait on the JS queue and dispatching them only when a certain deadline arrives, allowing for better usage of the computational resources.

Some algorithms may benefit from pipeline-oriented data processing algorithms, where a sequence of OpenCL kernels are applied in sequence to the same data set. Currently, this processing model is not supported by the framework, but we envisage that it should be possible to include it with only a limited set of changes in the source code.

The current implementation of the framework only supports submitting jobs to CPUs or GPUs. However, OpenCL has seen relevant developments since the first release of the specification, being supported by more and more manufacturers and by the release of new tools at a steady pace. OpenCL implementations are available for other multiprocessor architectures such as the IBM CellBE, FPGAs and some embedded devices [Zii]. MOSIX with VCL [BBNLS10] proposes to provide an OpenCL runtime in which all devices on a cluster of computers can be made visible as belonging to the same node. Our proposal does not impose any intrinsic limitations to the supported architectures. The framework can thus support the forthcoming runtimes by lightly tailoring the PU-M to better fit new OpenCL devices.

Not all OpenCL features are supported by the framework. For example, currently only global memory, read/write parameters can be associated with each job. The OpenCL

specification allows kernel arguments to be stored on a constant memory space, which can reduce kernel execution time. This issue should be addressed by our framework.

Due to the prototype and experimental nature of the current implementation, some features can still be optimized. The initial PU benchmarking kernels should be replaced by a more proper FLOPS assessment program. The Results Collector's RAM-only storage approach could also be extended to use other stable storage devices such as hard disks.

Very interesting speedups have been achieved by a relatively straightforward adaptation of existing programs to OpenCL and to the framework, without any further fine-grained optimizations. It is possible that these applications might achieve even greater speedups if these algorithms are fine-tuned or even partially rewritten so that they can take better advantage of the available hardware.

Other use-cases are planned to verify the performance gains achievable with other applications and different environments. Two interesting settings are the adaptation of an existing BLAS API to resort to the framework and a setting where multiple applications with different behaviours and requirements use the same pool of resources simultaneously, assessing the framework's workload-balancing capabilities under stress conditions.

Bibliography

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. "Linda and Friends". In: *Computer* 19.8 (Aug. 1986), pp. 26 –34. ISSN: 0018-9162. DOI: 10.1109/mc.1986.1663305.

[Amd67] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. Atlantic City, New Jersey: ACM, 1967, pp. 483–485. DOI: 10.1145/1465482.1465560.

[Esa] *ASAR Product Handbook*. European Space Agency. Feb. 2007.

[Atia] *ATI Stream Computing - Technical Overview*. AMD. 2009.

[Atib] *ATI Stream SDK v2.2*. http://developer.amd.com/gpu/AMDAPPSDK/downloads/pages/AMDAPPSDKDownloadArchive.aspx. 2010.

[AN08] C. Augonnet and R. Namyst. "A Unified Runtime System for Heterogeneous Multi-core Architectures". English. In: *2nd Workshop on Highly Parallel Processing on a Chip (HPPC 2008)*. Las Palmas de Gran Canaria Spain, 2008. URL: http://hal.inria.fr/inria-00326917/PDF/AugNam08HPPC.pdf.

[Aze09] R. Azevedo. "Sinthetic Aperture Radar Processor - ERS-2 Satellite". Diploma report. Portugal: Departamento de Informática da Universidade Nova de Lisboa, Sept. 2009.

[Bak00] M. Baker. "Cluster Computing White Paper". In: *CoRR* cs.DC/0004014 (2000). URL: http://arxiv.org/abs/cs.DC/0004014.

[BBNLS10] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. "A package for OpenCL based heterogeneous computing on clusters with many GPU devices". In: *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*. Sept. 2010, pp. 1 –7. DOI: 10.1109/clusterwksp.2010.5613086.

[BS10a] A. Barak and A. Shiloh. *MOSIX — User's and Administrator's Guides and Manuals*. Revised for MOSIX-2.29.0. 2010.

[BS10b] A. Barak and A. Shiloh. *The MOSIX Management System for Linux Clusters, Multi-Clusters, GPU Clusters and Clouds*. 2010.

[BN96] M. A. Beaumont and R. A. Nichols. "Evaluating Loci for Use in the Genetic Analysis of Population Structure". In: *Biological Sciences*. Vol. 263. 1377. The Royal Society, Dec. 1996, pp. 1619–1626.

[BFHSFHH04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. "Brook for GPUs: stream computing on graphics hardware". In: *ACM Trans. Graph.* 23.3 (2004), pp. 777–786. ISSN: 0730-0301. DOI: 10.1145/1015706.1015800.

[Cgm] *Cg Toolkit User's Manual*. Release 1.1. NVIDIA. Feb. 2003.

[Con] *Condor: High Throughput Computing*. http://www.cs.wisc.edu/condor/. 2010.

[Ocl] *Conformant Products — Khronos Group*. http://www.khronos.org/adopters/conformant-products#topencl. July 2010.

[Cor10] O. Corporation. *Java Remote Method Invocation Home*. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html. 2010.

[Cuda] *CUDA Community Showcase*. http://www.nvidia.co.uk/object/cuda_apps_flash_new_uk.html. July 2010.

[Nvi] *CUDA Toolkit 3.2 RC 2 — NVIDIA Developer Drivers for Linux*. http://developer.nvidia.com/object/cuda_3_2_toolkit_rc.html. Oct. 2010.

[Dij65] E. W. Dijkstra. "Solution of a problem in concurrent programming control". In: *Commun. ACM* 8.9 (1965), p. 569. ISSN: 0001-0782. DOI: 10.1145/365559.365617.

[DPaSMQO10] J. Duato, A. J. Peñ anda, F. Silla, R. Mayo, and E. S. Quintana-Orti. "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters". In: *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. July 2010, pp. 224–231. DOI: 10.1109/hpcs.2010.5547126.

[FQKYS04] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. "GPU Cluster for High Performance Computing". In: *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 47. ISBN: 0-7695-2153-3. DOI: 10.1109/sc.2004.26.

[FR95] D. G. Feitelson and L. Rudolph. "Parallel Job Scheduling: Issues and Approaches". In: *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, 1995, pp. 1–18. ISBN: 3-540-60153-8.

[FRS05] D. G. Feitelson, L. Rudolph, and U. Schwiegelshohn. "Parallel Job Scheduling – A Status Report". In: *Job Scheduling Strategies for Parallel Processing*. Vol. 3277/2005. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, pp. 1–16. ISBN: 978-3-540-25330-3. DOI: 10.1007/11407522_1.

[FRSSW97] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. C. Sevcik, and P. Wong. "Theory and Practice in Parallel Job Scheduling". In: *IPPS '97: Proceedings of the Job Scheduling Strategies for Parallel Processing*. London, UK: Springer-Verlag, 1997, pp. 1–34. ISBN: 3-540-63574-2.

[Fly72] M. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Trans. Comput.* C-21 (1972), pp. 948+. URL: http://en.wikipedia.org/wiki/Flynn's_taxonomy.

[FK99] I. Foster and C. Kesselman, eds. *The grid: blueprint for a new computing infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1-55860-475-8.

[FZRL08] I. Foster, Y. Zhao, I. Raicu, and S. Lu. "Cloud Computing and Grid Computing 360-Degree Compared". In: *Grid Computing Environments Workshop, 2008. GCE '08*. 2008, pp. 1 –10. DOI: 10.1109/gce.2008.4738445.

[Gee05] D. Geer. "Chip Makers Turn to Multicore Processors". In: *Computer* 38.5 (2005), pp. 11–13. ISSN: 0018-9162. DOI: 10.1109/mc.2005.160.

[GC92] D. Gelernter and N. Carriero. "Coordination languages and their significance". In: *Commun. ACM* 35.2 (1992), pp. 97–107. ISSN: 0001-0782. DOI: 10.1145/129630.129635.

[Pea] *"Google shivs server crowd with PeakStream buy"*. http://www.theregister.co.uk/2007/06/05/google_buys_peakstream/. June 2007.

[GSBCBL06] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. "Open MPI: A High-Performance, Heterogeneous MPI". In: *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*. Barcelona, Spain, 2006.

[Ora] *gridengine: Home*. http://gridengine.sunsource.net/. 2010.

[Gro08] O. Group. *CORBA 3.1*. http://www.omg.org/spec/CORBA/3.1/. Jan. 2008.

[HSSY00] V. Hamscher, U. Schwiegelshohn, A. Streit, and R. Yahyapour. "Evaluation of Job-Scheduling Strategies for Grid Computing". In: *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*. London, UK: Springer-Verlag, 2000, pp. 191–202. ISBN: 3-540-41403-7.

[Hed] *Hedeby: Hedeby Project Overview*. http://hedeby.sunsource.net/. 2010.

[Hen07] J. Hensley. "AMD CTM overview". In: *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*. San Diego, California: ACM, 2007, p. 7. DOI: 10.1145/1281500.1281648.

[HRFGA10] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard. "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations". English. In: *Europar 2010*. I.: Computing Methodologies/I.3: COMPUTER GRAPHICS, I.: Computing Methodologies/I.6: SIMULATION AND MODELING. Ischia-Naples Italy, Sept. 2010. URL: http://hal.inria.fr/inria-00502448/en/.

[Hpc] *HPC Wire: Global News and Information on High Performance Computing (HPC)*. http://www.hpcwire.com/. 2010.

[Hud91] R. R. Hudson. "Gene genealogies and the coalescent process". In: *Oxford Surveys in Evolutionary Biology* 7 (1991), 1–44.

[JVGGFN09] V. Jiménez, L. Vilanova, I. Gelado, M. Gil, G. Fursin, and N. Navarro. "Predictive Runtime Code Scheduling for Heterogeneous Architectures". In: *High Performance Embedded Architectures and Compilers*. Ed. by A. Seznec, J. Emer, M. O'Boyle, M. Martonosi, and T. Ungerer. Vol. 5409. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 19–33. DOI: 10.1007/978-3-540-92990-1_4.

[KKSEFAV92] E Katchalski-Katzir, I Shariv, M Eisenstein, A. A. Friesem, C Aflalo, and I. A. Vakser. "Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques". In: *Proceedings of the National Academy of Sciences of the United States of America* 89.6 (1992), pp. 2195–2199. eprint: http://www.pnas.org/content/89/6/2195.full.pdf+html. URL: http://www.pnas.org/content/89/6/2195.abstract.

[KBR09] J. Kessenich, D. Badwin, and R. Rost. *The OpenGL Shading Language*. Version 1.1. The Khronos Group. July 2009.

[KESSASPH09] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W. Hwu. "GPU Clusters for High-Performance Computing". In: *Proceedings of the Workshop on Parallel Programming on Accelerator Clusters (PPAC'09)*. Aug. 2009.

[Man83] B. B. Mandelbrot. *Fractals and the Geometry of Nature*. New York: Freeman, 1983.

[Mau] *Maui Administrator's Guide*. Maui 3.2. Supercluster Research and Development Group. May 2002.

[McC06] M. D. McCool. "Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform". In: *GSPx Multicore Applications Conference* (Nov. 2006).

[MDTPCM04] M. D. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. "Shader algebra". In: *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*. Los Angeles, California: ACM, 2004, pp. 787–795. DOI: 10.1145/1186562.1015801.

[Mab] *Moab Workload Manager — Administrator's Guide*. Version 5.4. Adaptive Computing Enterprises Inc. 2010.

[Mor03] R. S. Morrison. *Cluster Computing — Architectures, Operating Systems, Parallel Processing & Programming Languages*. Richard S. Morrison, 2003.

[Mpia] *MPI: A Message-Passing Interface Standard*. Version 2.2. Message Passing Interface Forum. Sept. 2009.

[Mpib] *MPICH2 : High-performance and Widely Portable MPI*. http://www.mcs.anl.gov/research/projects/mpich2/. 2010.

[Mun08] A. Munshi. *OpenCL — Parallel Computing on the GPU and CPU*. SIGGRAPH — Beyond Pragammable Shading. 2008.

[Mun10] A. Munshi. *The OpenCL Specification*. 1.1.36. 2010.

[Cudb] *NVIDIA CUDA Programming Guide*. Version 2.3.1. NVIDIA. Aug. 2009.

[OH05] K. Olukotun and L. Hammond. "The Future of Microprocessors". In: *Queue* 3.7 (2005), pp. 26–29. ISSN: 1542-7730. DOI: 10.1145/1095408.1095418.

[Ope] *Open Grid Forum*. http://www.ogf.org. 2010.

[Omp] *OpenMP Application Program Interface*. 3.0. OpenMP Architecture Review Board. May 2008.

[OHLGSP08] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. "GPU Computing". In: *Proceedings of the IEEE* 96.5 (May 2008), pp. 879–899. DOI: 10.1109/JPROC.2008.917757. URL: http://graphics.idav.ucdavis.edu/publications/print_pub?pub_id=936.

[OLGHKLP07] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware". In: *Computer Graphics Forum* 26.1 (2007), pp. 80–113. URL: http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x.

[Pap07] M. Papakipos. *The PeakStream Platform: High Productivity Software Development for Multi-Core Processors*. May 2007. URL: http://www.linuxclustersinstitute.org/conferences/archive/2007/PDF/papakipos_21367.pdf.

[Rap] *"RapidMind is now part of Intel Corporation"*. http://software.intel.com/en-us/blogs/2009/08/19/rapidmind-intel/. Aug. 2009.

[SBF94] B. R. Seyfarth, J. L. Bickham, and M. R. Fernandez. "Glenda: an environment for easy parallel programming". In: *Scalable High-Performance Computing Conference, 1994., Proceedings of the*. 1994, pp. 637 –641. DOI: 10.1109/shpcc.1994.296701.

[SMV10] K. Spafford, J. Meredith, and J. Vetter. "Maestro: data orchestration and tuning for OpenCL devices". In: *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*. Euro-Par'10. Ischia, Italy: Springer-Verlag, 2010, pp. 275–286. ISBN: 3-642-15290-2, 978-3-642-15290-0. URL: http://portal.acm.org/citation.cfm?id=1885276.1885305.

[Sto09] J. E. Stone. *Introduction to OpenCL Webinar*. http://www.gpucomputing.net/?q=node/128. Dec. 2009.

[SKSS02] V. Subramani, R. Kettimuthu, S. Srinivasan, and P. Sadayappan. "Distributed job scheduling on computational Grids using multiple simultaneous requests". In: *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing* (Nov. 2002), pp. 359 –366.

[TC00] X. Tang and S. T. Chanson. "Optimizing Static Job Scheduling in a Network of Heterogeneous Computers". In: *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*. Washington, DC, USA: IEEE Computer Society, 2000, p. 373. ISBN: 0-7695-0768-9.

[TPO06] D. Tarditi, S. Puri, and J. Oglesby. "Accelerator: using data parallelism to program GPUs for general-purpose uses". In: *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. San Jose, California, USA: ACM, 2006, pp. 325–335. ISBN: 1-59593-451-0. DOI: 10.1145/1168857.1168898.

[Ber] *The Berkeley NOW Project*. http://now.cs.berkeley.edu. 2010.

[Pth] *The Open Group Base Specifications Issue 6 — IEEE Std 1003.1, 2004 Edition — pthread.h*. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html. 2004.

[Tse] *The Open Group Base Specifications Issue 6 — IEEE Std 1003.1, 2004 Edition — tdelete, tfind, tsearch, twalk - manage a binary search tree*. http://pubs.opengroup.org/onlinepubs/009695399/functions/tsearch.html. 2004.

[Pbs] *The Portable Batch System*. http://www.nas.nasa.gov/Software/PBS/. 1998.

[Thu09] R. Thurlow. *RPC: Remote Procedure Call Protocol Specification Version 2*. http://tools.ietf.org/html/rfc5531. May 2009.

[Top] *TOP500 Supercomputing Sites*. http://www.top500.org/.

[Tor] *Torque Administrator's Manual*. Version 2.4.5. Adaptive Computing Enterprises, Inc. 2010.

[WC84] B. S. Weir and C. C. Cockerham. "Estimating F-Statistics for the Analysis of Population Structure". In: *Evolution* 38.6 (Nov. 1984), pp. 1358–1370.

[Hls] *Windows DirectX Graphics Documentation — HLSL*. http://msdn.microsoft.com/en-us/library/ee418149%28VS.85%29.aspx. Microsoft, Aug. 2009.

[Wri43] S. Wright. "Isolation by distance". In: *Genetics Soc America* (1943).

[YS09] S. Yamagiwa and L. Sousa. "CaravelaMPI: Message Passing Interface for Parallel GPU-Based Applications". In: *ISPDC '09: Proceedings of the 2009 Eighth International Symposium on Parallel and Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 161–168. ISBN: 978-0-7695-3680-4. DOI: 10.1109/ispdc.2009.24.

[Zii] *ZiiLABS OpenCL SDK Preview Access Program*. http://www.ziilabs.com/opencl. June 2010.

API

This Appendix describes the API services made available to simplify the integration of applications with the current framework. Some familiarity with the OpenCL platform, execution and memory models is required in order to understand the concepts and technologies on which this API depends. Refer to Section 2.4.4 of this document and Section 3 of the OpenCL Specification [Mun10] for a complete overview and description of the core principles behind this standard. Some familiarity with the MPI Standard [Mpia] is recommended, although not required.

Currently, the API provides data types, data structures and functions in the C programming language. The purpose of these services is to ease the implementation of a Job Manager. All of these services can be overridden by the Job Manager. However, this implies that the user of the framework must also be concerned with the correctness of the interface between the application and the remaining components of the framework. By resorting to this API, the user is released from this burden and can focus on the implementation of the Job Manager. Refer to Chapter 3 for a detailed description of the overall system and each of its components.

In order to submit a job or a set of jobs to the framework for execution, some API function calls are mandatory, others are optional while others are mutually exclusive. A careful read of these pages is recommended before attempting to use any of these API functions.

A.1 API Data Types, Data Structures and Global Variables

Data Types

All OpenCL runtime data types (**cl_int**, **cl_char**, **cl_float**, ...) are available to the Job Manager. All arguments associated with jobs to be submitted to the framework must be of one of these types.

Data Structures

The API provides two data structures to the Job Manager: **Job** and **JobResults**. **Job**s should not be manipulated manually. Their fields should only be modified indirectly by API functions. Non-compliance may result in unspecified behaviour. In order to read the contents of **JobResults** data structures' fields, the user may access its fields directly. However, they should not be modified. Listing A.1 presents the relevant fields of the **JobResults** data structure.

```
typedef struct {
  (...)
  int jobID;

  int nTotalResults;
  int *resultSizes;
  void **results;

  int returnStatus;
} JobResults;
```

Listing A.1: The relevant fields of the JobResults data structure

Listing A.2 presents an example covering the typical process for retrieving data from a **JobResults** data structure. In this example, a single job was submitted and only one output argument was associated with the job. Hence, there is only one result buffer to be fetched.

Enumerations

The admissible types for the jobs' arguments are defined on the **argument_type** enumeration. These are the types required by the `setArgument` API call and are shown in Listing A.3.

The `set*PU` API functions require an OpenCL device type to be indicated. Currently, only the OpenCL-defined CL_DEVICE_TYPE_CPU and CL_DEVICE_TYPE_GPU values of the **cl_device_type** enumeration are supported.

```
Job job;
int *localResultBuffer;

void resultAvailable (int jobID) {
  JobResults *JR = getResults(job);


  if (JR == NULL) {
    fprintf(stderr,"An error has occurred while fetching the results for the
        job with ID %i.\n", jobID);
    //cleanup
    return;
  }
  printf("Results successfully fetched for Job with ID %i.\n", jobID);
  printf("Job execution return status: ");
  switch(JR->returnStatus) {
    case (JOB_RETURN_STATUS_SUCCESS):
      printf("kernel executed successfully.\n");
      break;
    case default:
      fprintf(stderr,"kernel failed while executing (error %i). Please check
          the system log.\n", JR->returnStatus);
      //cleanup
      return;
  }
  assert(JR->nTotalResults == 1);
  localResultBuffer = malloc(JR->resultSizes[0]);
  memcpy(localResultBuffer, JR->results[0], JR->resultSizes[0]);
  deleteResults(JR);
  printf("First value on the result buffer: %i\n",localResultBuffer[0]);
}
```

Listing A.2: Sample implementation of the resultAvailable callback function.

```
typedef enum {
  INPUT,
  OUTPUT,
  INPUT_OUTPUT,
  EMPTY_BUFFER
} argument_type;
```

Listing A.3: The argument_type enumeration

Global Variables

The following global variables are available to the Job Manager:

int defaultSchedID — The Unique Identifier for the Job Sechduler.

int defaultRCID — The Unique Identifier for the Results Collector.

These variables can be assigned automatically by resorting to the initDistribCL API call and their values can be queried (but may not be modified) by the Job Manager.

A.2 API Functions

void initDistribCL (int argc, char *argv[])

int argc — Argument count

char *argv[] — Argument vector

Initializes the environment and performs any necessary, initial interactions with the other components of the framework.

`argv` may not contain arguments other than `--job-scheduler` and `--result-collector`, each followed by a number, indicating the MPI rank of each of these components. By specifying these ranks, the `defaultSchedID` and `defaultRCID` global variables are available to the programmer, and may be used in subsequent calls to identify these components.

This function must be called before all other API functions.

void quitDistribCL (void)

Notifies the framework that no more operations will be requested from it. The Job Manager may continue execution, but no further calls to API functions shall be made afterwards.

Job *createJob (void)

Creates a new, empty job. The attributes of this job must be provided with subsequent calls to API functions.

return — A pointer to a newly-created job.

void setJobID (Job *job, int jobID)

Job *job — A previously-created job (see `createJob`)

int jobID — A numerical identifier

Assigns an identifier to a job. This must be an unique identifier and may not be reused until the job results are returned to the sender. Reusing an identifier is illegal and

the resulting behaviour is unspecified.

This function must be called before a job is requested to execute.

void setRequiredPU (Job *job, cl_device_type requirePU)

Job *job	— A previously-created job (see `createJob`)
cl_device_type requirePU	— An OpenCL device type where the job is required to be processed on

A given job's task may be designed in such a way that it has to run on a certain type of Processing Unit. If this is the case, `setRequiredPU` should be called, identifying the (OpenCL) type of such hardware.

After calling this function, the remaining `set*PU` API functions may not be called for the same job.

void setPreferPU (Job *job, cl_device_type preferPU)

Job *job	— A previously-created job (see `createJob`)
cl_device_type preferPU	— An OpenCL device type where the job is preferred to be processed on

A given job's task may be designed in such a way that it is predictable it will perform better on some types of Processing Unit than on others. `setPreferPU` may be called, by order of preference to indicate this preference.

If `setRequiredPU` was not used, at least this or one of the remaining `set*PU` API functions must be called at least once so that the job may execute on an available Processing Unit.

void setAllowPU (Job *job, cl_device_type allowPU)

Job *job	— A previously-created job (see `createJob`)
cl_device_type allowPU	— An OpenCL device type where the job is allowed to be processed on

If a given type of Processing Unit should be capable of executing a given job, that should be indicated using this function.

If `setRequiredPU` was not used, at least this or one of the remaining `set*PU` API functions must be called at least once so that the job may execute on an available Processing Unit.

void setAvoidPU (Job *job, cl_device_type avoidPU)

Job *job	— A previously-created job (see `createJob`)
cl_device_type avoidPU	— An OpenCL device type where the framework must avoid processing the job on

Although they could run on a given type of Processing Unit, certain jobs should not run on certain PU types. For example, jobs that can only be executed in parallel by a small number of simultaneous OpenCL global items should avoid being run on a GPU. Such restrictions should be indicated by resorting to the `setAvoidPU` function.

If `setRequiredPU` was not used, at least this or one of the remaining `set*PU` API functions must be called at least once so that the job may execute on an available Processing Unit.

void setForbidPU (Job *job, cl_device_type forbidPU)

Job *job	— A previously-created job (see `createJob`)
cl_device_type forbidPU	— An OpenCL device type where the job must not be processed

Certain device types are not indicated for performing certain kinds of jobs. For example, a long-lasting job that may not run in parallel or that may only be processed by a single OpenCL item, must not be let run on a GPU. These restrictions should be indicated by resorting to the `setForbidPU` function.

void setJobCategory (Job *job, int category)

Job *job	— A previously-created job (see `createJob`)
int category	— A user-defined category for this job

Jobs may be categorized based on their characteristics (see Section 3.2.1). Depending on the selected scheduling algorithm for the Job Scheduler, more adequate scheduling decisions can be made when job categories are cleverly chosen. Refer to Section 3.3.4 for a reference on the currently implemented scheduling algorithms that take job categories into account.

Calling this function is optional. Although jobs may be submitted without any associated category, using this facility may significantly improve overall Job turnaround times (see Chapter 4 for an experimental assessment of such improvements for different applications).

void setDimensions (Job *job, int nDim, int *nItemsPerDim, int *nItemsPerGroup)

Job *job	— A previously-created job (see `createJob`)
int nDim	— The number of dimensions for the OpenCL index-space (NDRange) that the job's task will be mapped to
int *nItemsPerDim	— The number of global work-items on each dimension that the job's task will be mapped to
int *nItemsPerGroup	— The number of local work-items on each dimension that the job's task will be mapped to

This function associates an OpenCL NDRange index space configuration (see Section 2.4.4) to the given job's task. This configuration usually has a significant impact on a job's execution time and as such it must be chosen with care.

Calling this function is optional. If it is not called for a given job, the following configuration is used: the index space of the task will be one-dimensional; the number of global items will be the maximum supported by the OpenCL device where the job is executed; and the number of items per group will be determined in run-time by the supporting OpenCL implementation. Leaving this decision to the runtime may lead to sub-optimal execution times.

void loadSourceFile (Job *job, char *taskSourceFile)

Job *job	— A previously-created job (see `createJob`)
char *taskSourceFile	— The pathname of an OpenCL source file

Loads the contents of a file stored on the file system onto the job's task. This file's contents will be the source code of the OpenCL kernel that shall be executed upon the submission of the job. `taskSourceFile` is expected to be a null-terminated string indicating the pathname of the file. Currently, only a single source file per job is supported.

Calling this function is mandatory before submitting a job for execution.

void setStartingKernel (Job *job, char *startingKernel)

Job *job	— A previously-created job (see `createJob`)
char *startingKernel	— The name of the starting kernel

The OpenCL source file may contain multiple function definitions. `setStartingKernel` must be used to identify the name of the function where the task's execution shall start. This starting function can be seen as the analogous to the `main` function on a standard C program.

Calling this function is mandatory before submitting a job for execution.

void setArgument (Job *job, argument_type argType, size_t argSize, void *argument)

Job *job	— A previously-created job (see `createJob`)
argument_type argType	— The type of the argument
size_t argSize	— Size of the argument, in bytes
void *argument	— A pointer to the argument's data

Associates data with a job. Each time `setArgument` is called, the data pointed to by `argument` is associated with the corresponding parameter of the job task's starting function. The first time `setArgument` is called, it associates the data to the first parameter.

The second time, to the second parameter, and so forth. The order by which data is associated with a job must correspond to the order of the arguments of the starting function's prototype. `setArgument` must be called as many times as there are arguments for this starting function.

The `argType` parameter identifies the *type* of the current argument. Currently this can be one of: INPUT, OUTPUT, INPUT_OUTPUT or EMPTY_BUFFER. A description of each follows:

- INPUT — data that is going to be made available, unmodified, to the kernel as one of it's arguments. This data will be discarded upon job execution completion;
- OUTPUT — a memory space that will return data when the job's execution is complete. Its contents are to be filled by the running kernel;
- INPUT_OUTPUT — data that is both going to be made available to the running kernel as well as returned to the Job Manager after the job's execution is over;
- EMPTY_BUFFER — a memory space that is going to be allocated before the job's execution starts and used as a kernel argument. Its contents are to be filled by the running kernel. This data is discarded upon job execution completion.

When calling `setArgument`, INPUT and INPUT_OUTPUT argument types need to be allocated and point to valid memory locations. This data must remain consistent until the job is sent for execution. `free()`ing this data is also the responsibility of the Job Manager implementation. OUTPUT and EMPTY_BUFFER argument types need not to point to a valid memory location (i.e., a `NULL argument` can be used) — but an adequate corresponding `argSize` is mandatory.

As of the current version, all arguments are associated with the OpenCL device's global memory space, as read-write buffers. This means that all of the task's starting function's parameters must be declared with the `__global` qualifier. It might also be desirable to take this into account in order to implement performance adaptations at the kernel level.

A job may not be submitted for execution without having at least one argument associated with it.

void setResultsCollector (Job *job, int rcID)

Job *job — A previously-created job (see `createJob`)
int rcID — A Results Collector's MPI Rank

Associates a job with a Results Collector. This will be the RC where the job's output data will be available after its execution is complete. If the `--result-collector` argument is provided to the `initDistribCL` API call, the `defaultRCID` global variable

may be used as this function's `rcID` argument.

A job may not be submitted to execution without having been associated with a Results Collector.

void requestResultNotification (Job *job)

Job *job — A previously-created job

Request a notification on the availability of the output data for this job from the associated Results Collector. After calling this function, the RC will notify the Job Manager after the job's execution is complete and the resulting output data is available to be fetched. This notification is signaled to the Job Manager by the invocation of the callback function `resultAvailable`. This function must be implemented by the Job Manager, independently of `requestResultNotification` being called or not.

Calling this function is optional. It may be called either before or after the job was submitted for execution.

The callback function's prototype is as follows: **void** resultAvailable (**int** jobID), where jobID corresponds to the ID of the submitted job for which the resulting data is available. This callback function is invoked on a new thread of the Job Manager's process. Thread-safety must therefore be taken into account when implementing this function.

void sendJobToExec (Job *job, int schedID)

Job *job — A previously-created and configured job
int schedID — A Job Scheduler's MPI Rank

This function submits the job to the provided Job Scheduler so that it can be executed in one of the system's Processing Units. If the `--job-scheduler` argument is provided to the `initDistribCL` API call, the `defaultSchedID` global variable may be used here.

This function may only be called if all of the following API functions were already called for the given job: `createJob`, `setJobID`, `setRequiredPU` or at least of the other `set*PU` functions, `loadSourceFile`, `setStartingKernel`, `setArgument` and `setResultsCollector`. It may not be called after `deleteJob` was been called for the same job.

JobResults *getResults (Job *job)

Job *job — A previously-created and configured job

Fetches the output data of the given job, if it has already completed execution. If the results are not available yet, a `NULL` value is returned.

The data inside a **JobResults** data structure may be read directly but should not be changed manually. If changing this data is required, it should first be copied to a buffer local to the JM.

return — A pointer to a **JobResults** data structure, containing the output data of the given job. `NULL` if the results were not available at the time of the invocation.

void deleteJob (Job *job)

Job *job — A previously-created job

Deletes a job, freeing all resources allocated to it. The arguments that may have been associated with the job are not freed and their management is the responsibility of the Job Manager's implementation.

After calling this function, no more operations or API functions may be called over the same job: it has been definitely erased from memory.

void deleteResults (JobResults *JR)

JobResults *JR — A **JobResults** data structure returned by the `getResults` API function

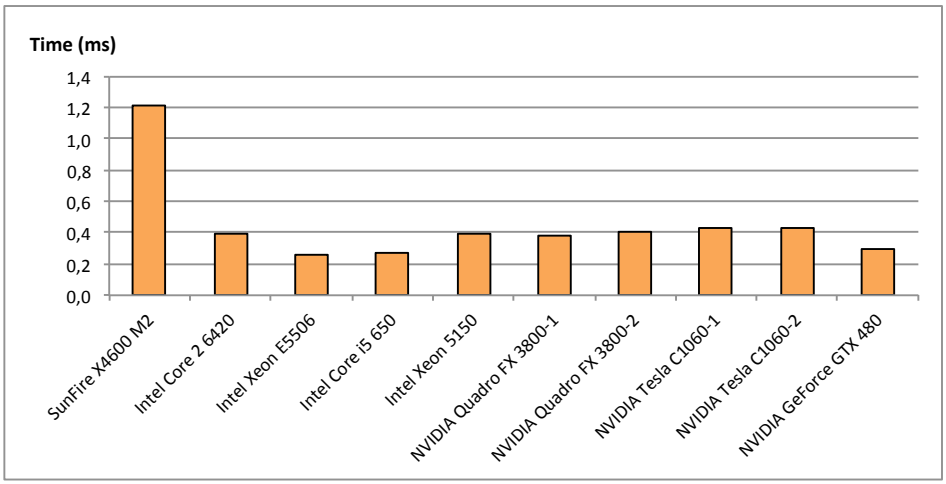
Deletes a **JobResults** data structure, freeing all resources allocated to it. This structure is expected to have been returned by the `getResults` API call. Calling `deleteResults` over data structures created otherwise results in unspecified behaviour.

After calling this function, no more operations may be called over the same structure: it has been definitely erased from memory.

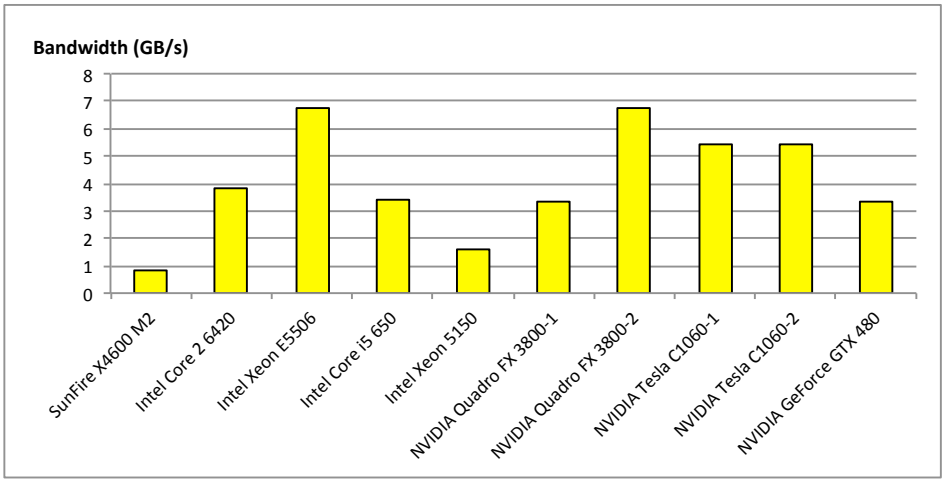
B

Additional Charts

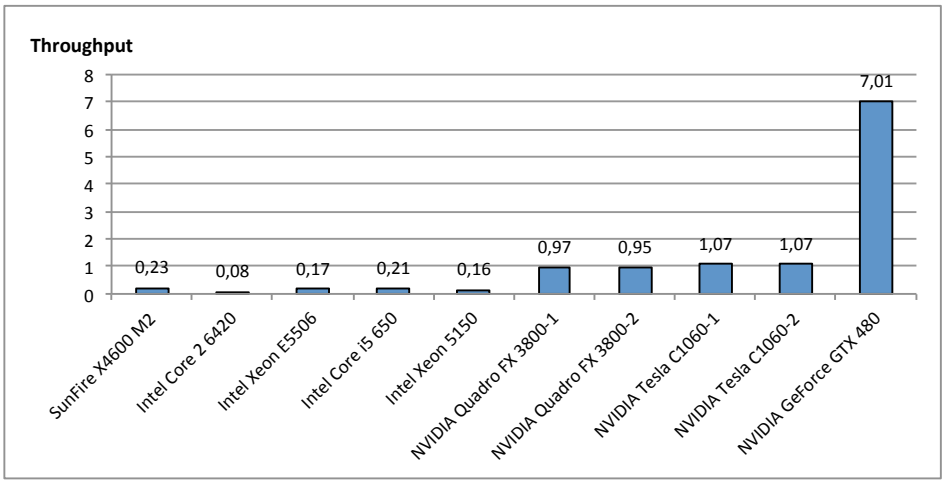
This Appendix provides additional figures with more detailed versions of some of the charts that are presented and described on the Evaluation Chapter. The PU Properties inferred from the Processing Unit Managers installed on the various machines presented in Section 4.2 are provided with a higher detail. These charts are followed by more detailed views of the turnaround times registered by the Job Manager for the local, configuration tests as well as the distributed benchmarks, described in Section 4.4.



(a) Kernel submission latency.

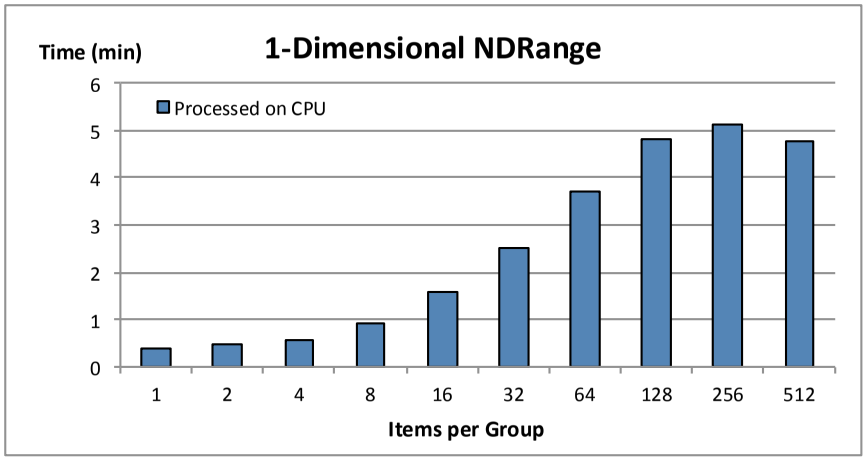


(b) Host-to-Device data transfer bandwidth.

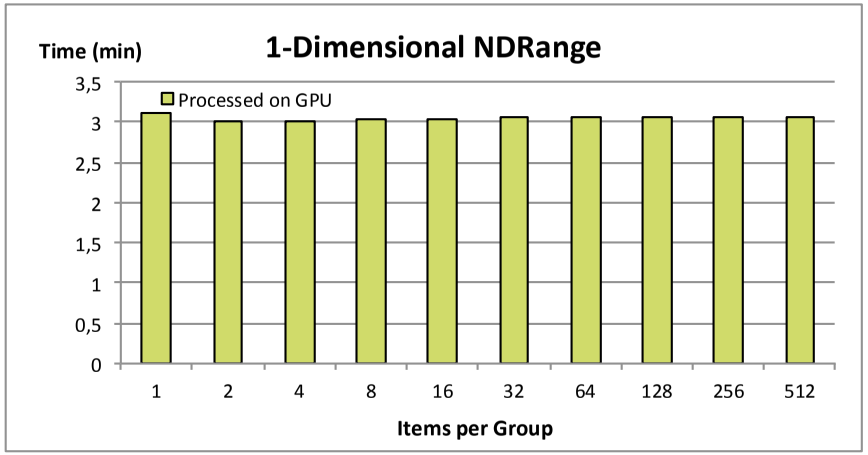


(c) PU parallel processing throughput.

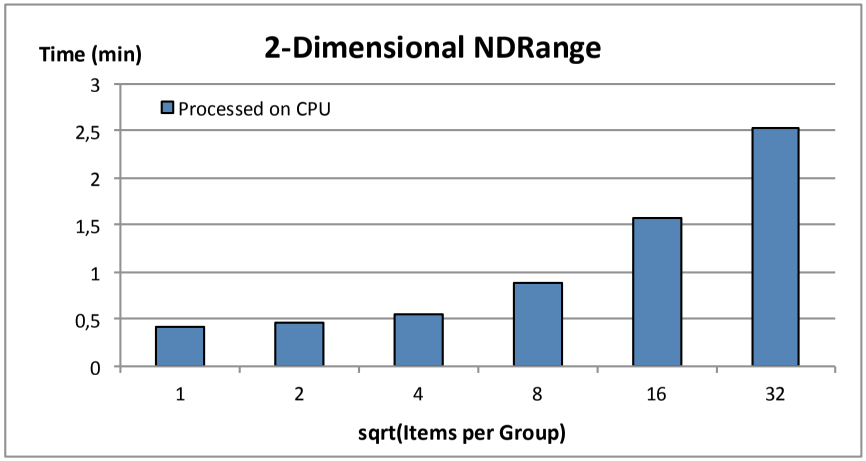
Figure B.1: Device properties as inferred by the Processing Unit Managers.



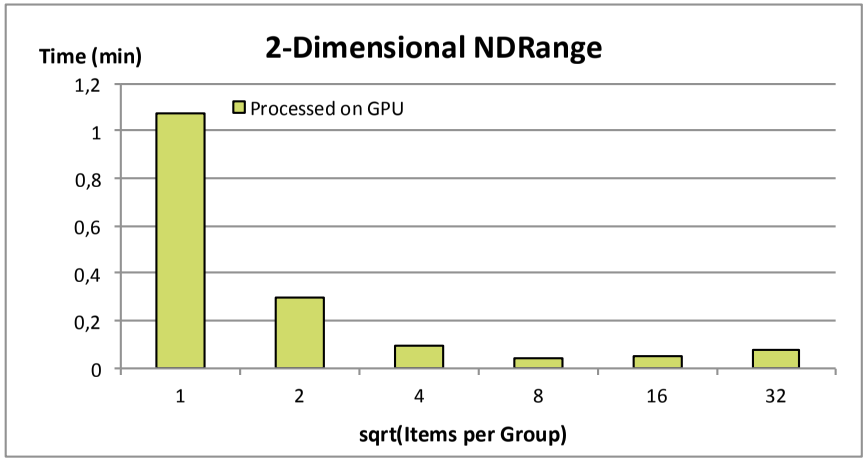
(a) Mandelbrot Set, CPUs, 1-dimensional NDRange.



(b) Mandelbrot Set, GPUs, 1-dimensional NDRange.

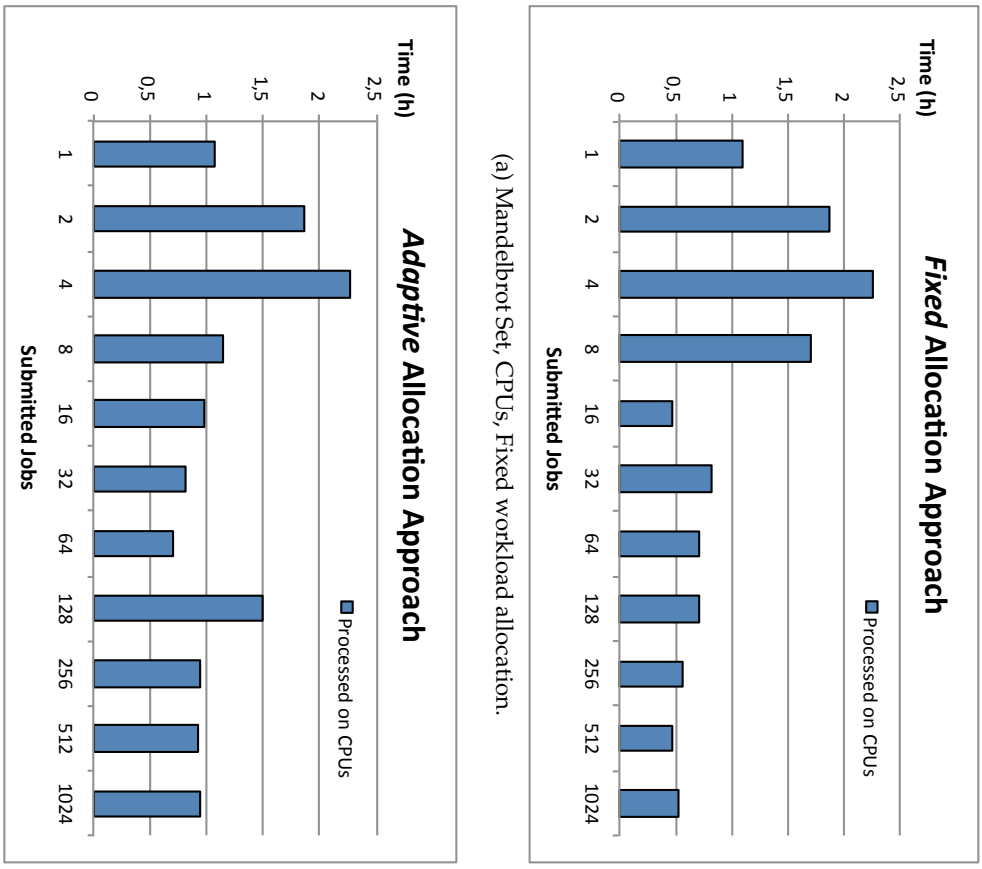


(c) Mandelbrot Set, CPUs, 2-dimensional NDRange.

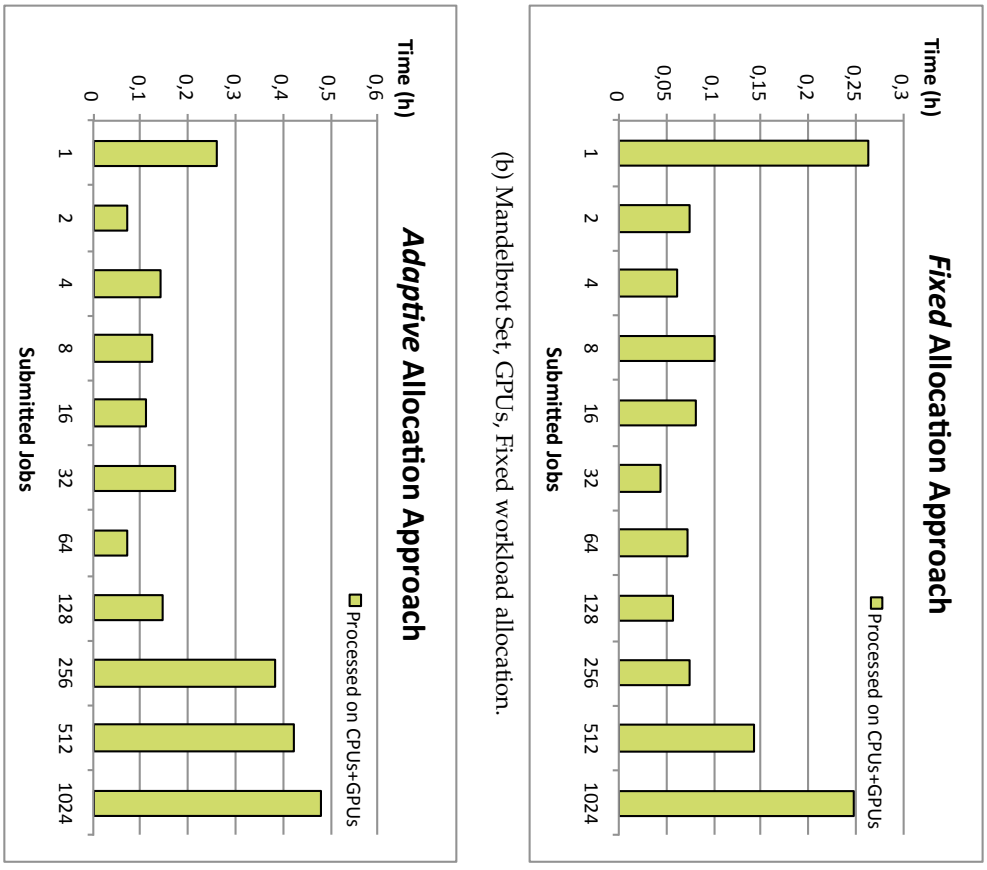


(d) Mandelbrot Set, GPUs, 2-dimensional NDRange.

Figure B.2: Turnaround time for the OpenCL (local) Mandelbrot Set kernel.



(a) Mandelbrot Set, CPUs, Fixed workload allocation.

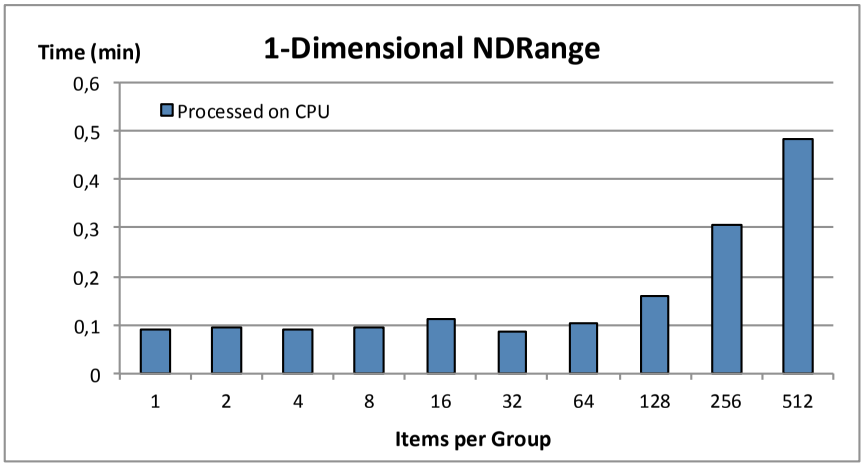


(b) Mandelbrot Set, GPUs, Fixed workload allocation.

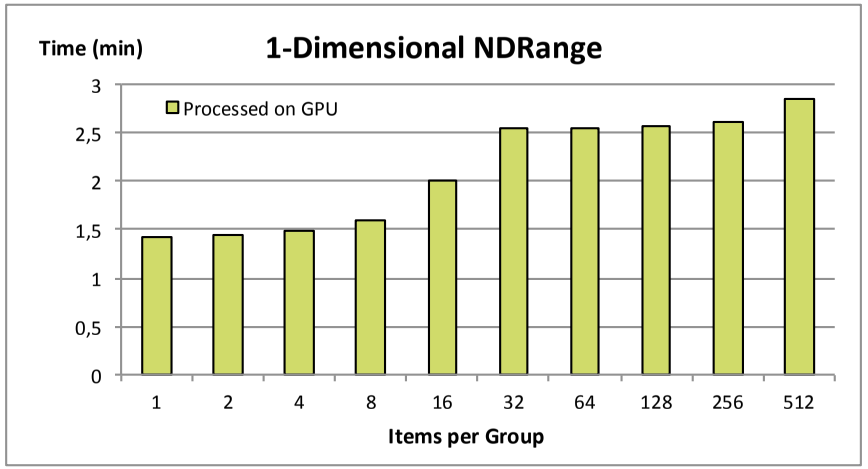
(c) Mandelbrot Set, CPUs, Adaptive workload allocation.

(d) Mandelbrot Set, GPUs, Adaptive workload allocation.

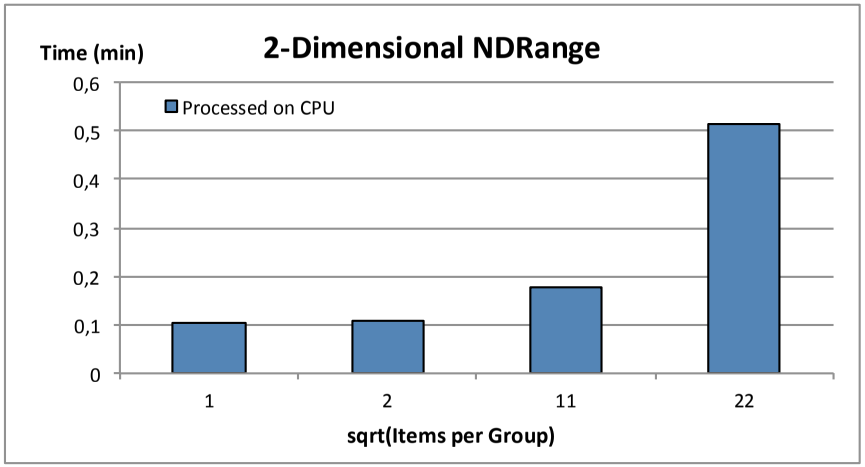
Figure B.3: Turnaround time for the distributed Mandelbrot Set computation.



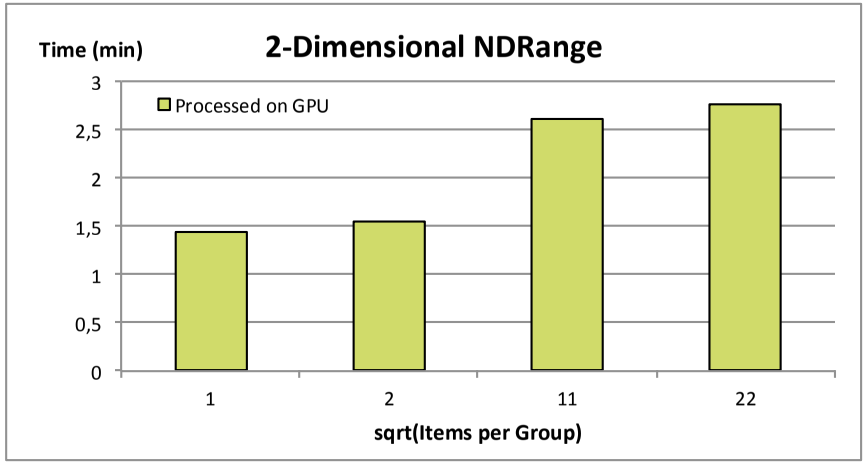
(a) Gene identification, CPUs, 1-dimensional NDRange.



(b) Gene identification, GPUs, 1-dimensional NDRange.

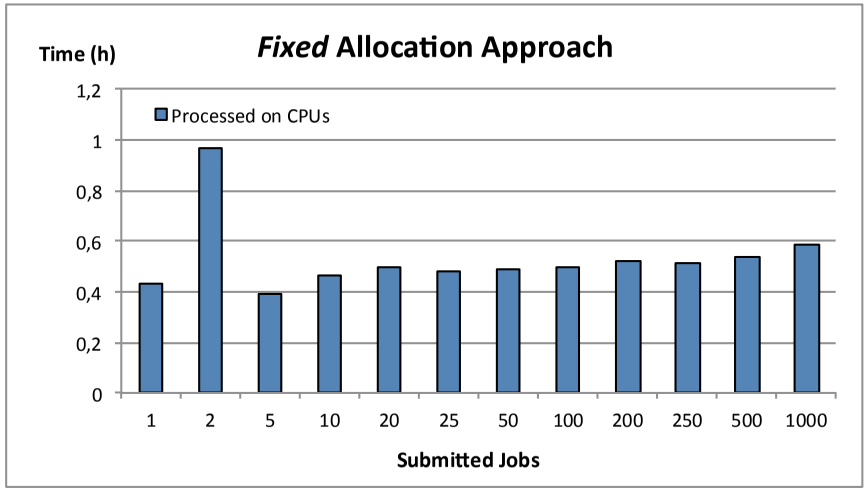


(c) Gene identification, CPUs, 2-dimensional NDRange.

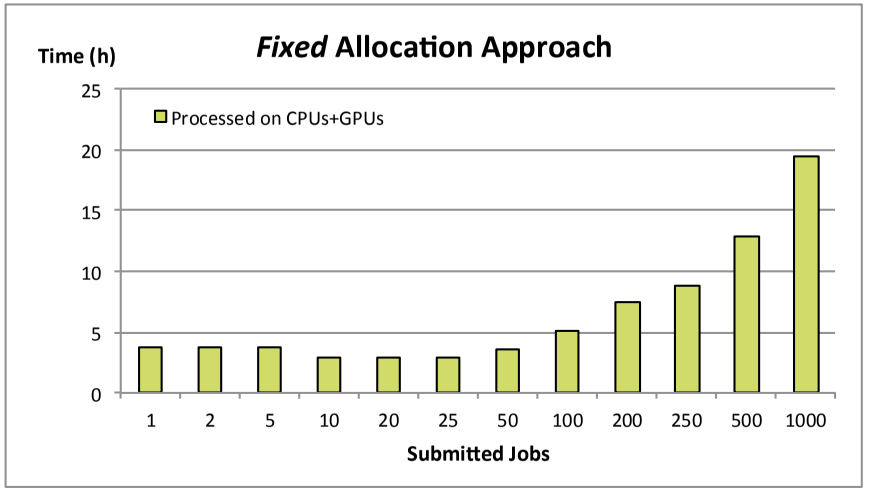


(d) Gene identification, GPUs, 2-dimensional NDRange.

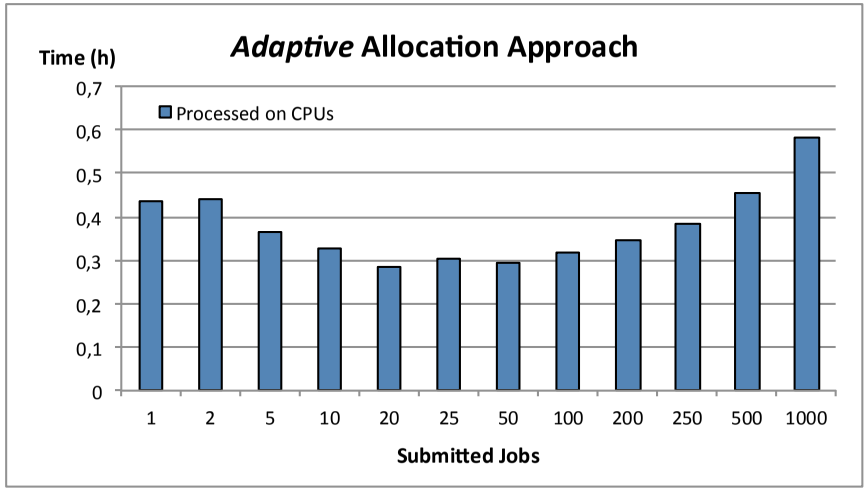
Figure B.4: Turnaround time for the OpenCL (local) Gene identification kernel.



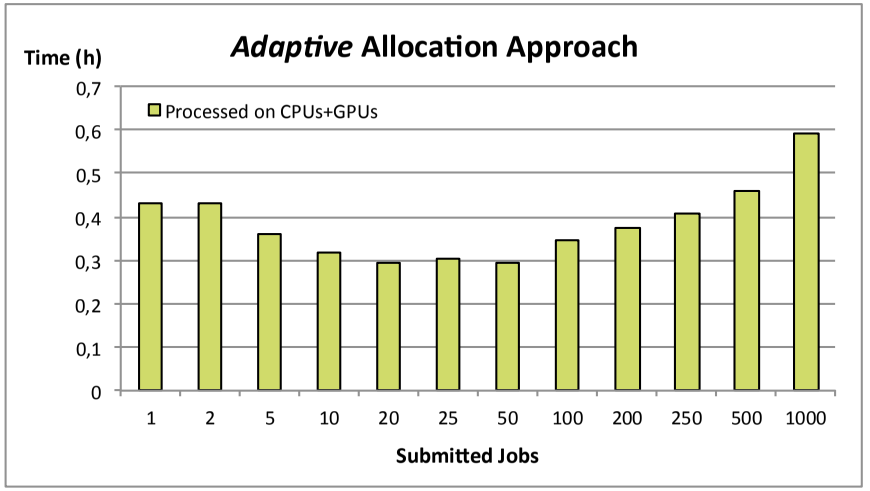
(a) Gene identification, CPUs, 1-dimensional NDRange.



(b) Gene identification, GPUs, 1-dimensional NDRange.



(c) Gene identification, CPUs, 2-dimensional NDRange.



(d) Gene identification, GPUs, 2-dimensional NDRange.

Figure B.5: Turnaround time for the distributed Gene identification kernel.